THE APPLICATION PERSPECTIVE OF MUTATION TESTING

QIANQIAN ZHU

THE APPLICATION PERSPECTIVE OF MUTATION TESTING

THE APPLICATION PERSPECTIVE OF MUTATION TESTING

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op woensdag 24 juni 2020 om 12:30 uur

door

Qianqian ZHU

Master of Science in Computer Science,
Imperial College London, Verenigd Koningkrijk,
geboren te Wenzhou, China.

The work in the thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics) and was financed by the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (NWO), project TestRoots, grant number 016.133.324.

Keywords: Software Testing, Mutation Testing, Software Quality...

Printed by: Gildeprint

Front & Back: Beautiful cover art that captures the entire content of this thesis in a single illustration.

ISBN 000-00-0000-000-0

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

CONTENTS

SUMMARY

The main goal of this thesis is to investigate, improve and extend the applicability of mutation testing. To seek the potential directions of how to improve and extend the applicability of mutation testing, we have started with a systematic literature review on the current state of how mutation testing is applied. The results from the systematic literature review have further guided us towards three directions of research: (1) speeding up mutation testing; (2) deepening our understanding of mutation testing; (3) exploring new application domains of mutation testing.

For the first direction, we have leveraged compression techniques and weak mutation information to speed up mutation testing. The results have shown our proposed mutant compression techniques can effectively speed up strong mutation testing up to 94.3 times with an accuracy > 90%. Given the second direction, we are interested in gaining a better understanding of mutation testing especially in the situation where engineers cannot kill *all* the mutants by just adding test cases. We have investigated the relationships between code quality regarding the testability and observability, and the mutation score. We have observed a correlation between observability metrics and the mutation score. Furthermore, relatively simple refactoring operations/adding tests enable an increase in the mutation score.

As for the third direction, we have explored two new application domains: one is physical computing, and the other is GPU programming. In both application domains, we have designed new mutation operators based on our observations of the common mistakes that could happen during the implementation of the software. We have found promising results in that mutation testing can help in revealing weaknesses of the test suite for both application domains.

In summary, we have improved the applicability of mutation by proposing a new speed-up approach and investigating the relationship between testability/observability and mutation testing. Also, we have extended the applicability of mutation testing in physical computing and GPU programming domains.

Acknowledgements

First of all, I would like to thank my supervisor and promotor Prof. Dr. Andy Zaidman for his great support during my 4-year PhD research. Without his offer, I would never have this opportunity to start a PhD journey. Also, despite his busy schedule, he is always willing to answer my endless questions and give me numerous helpful suggestions during the four years. He is unstinting in his expertise and experience to guide me: to set my topic, to frame each of my chapters and to work towards this thesis in a structured way. In the meanwhile, he encouraged me and praised me for every progress I made for my research.

I am also greatly indebted to my family who supports my oversea study not only financially but also mentally. They never stop encouraging me and cheering me up from the distance. Thank you is not enough for my husband, Ce, who always supports me unconditionally and listens to my endless nagging.

Finally, I would like to thank all my friends and colleagues for accompanying me and providing great help during my PhD research. Without them, the four years would not have been so colourfully and unforgettable.

1 INTRODUCTION

Software testing has always been a hot topic in the research field of software engineering [77]. The critical goal of software testing is to ensure high quality and reliable software programs. In 1972, Dijkstra made the well-known statement that “program testing can be used to show the presence of bugs, but never to show their absence" [124]. His saying indicates that testing is not the silver bullet for software quality. Later, Goodenough and Gerhart [159] proved that *properly* structured tests are capable of demonstrating the absence of errors in a program. This finding opened up the new research domain of “test criteria" [386]. Various test criteria have been proposed and studied over the last four decades, such as statement coverage, branch coverage and *mutation adequacy* [386]. The first two criteria, statement coverage and branch coverage, are both under the category of *structural coverage criteria*; this category specifies testing requirements in terms of the *coverage* of a particular set of elements in the structure of the program or the specification [386]. Different from structural coverage criteria, *mutation adequacy* is introduced by *mutation testing* (or mutation analysis) [117, 168, 230], a *fault-based testing* technique that assesses the test suite quality by systematically introducing small artificial faults [197]. The mutation adequacy score is used to measure the effectiveness of a test set in terms of its ability to detect faults.

Mutation testing originated in the 1970s with works from Lipton [230], DeMillo et al. [117] and Hamlet [168] and has been a very active research field over the last few decades. The activeness of the field is in part evidenced by the extensive survey of more than 390 papers on mutation testing that Jia and Harman published in 2011 [197]. Jia and Harman's survey highlights the research achievements that have been made over the years, including theories, problems, cost reduction techniques, applications, empirical evaluation, and tools [197]. This literature review was later extended by Papadakis et al. [289] with more recent advances in mutation testing.

While existing surveys (e.g., [197, 241, 279]) provide us with a great overview of the most influential realisations in research, we lack insight into how mutation testing is actually *applied*. This thesis focuses on the *applicability* of mutation testing. We first aim to understand the current state of how mutation testing is applied in a research context, thereby not excluding industrial practice, and then improve and extend the applicability of mutation testing.

1.1. MUTATION TESTING

The idea behind mutation testing is based on two fundamental hypotheses: the *Competent Programmer Hypothesis* [117] and the *Coupling Effect* [117, 268]. DeMillo et al. [117] introduced the Competent Programmer Hypothesis in 1978: "*The competent programmers create programs that are close to being correct.*" This hypothesis implies that the potential faults in the programs delivered by the competent programmers are just very simple mistakes; these defects can be corrected by a few simple syntactical changes. Thereby, mutation testing typically applies small syntactical changes to original programs to resemble faults made by "competent programmers".

The *Coupling Effect* further strengthens the first hypothesis by stating: "*Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors*" [117]. This means complex faults are *coupled* to simple faults. As a result, the mutants used in traditional mutation testing are only limited to first-order mutants, which are created by applying the mutation operator to the original program once.

Figure 1.1 summarises the *generic process* of mutation testing: *given a program P and a test suite T, first run T on P (Step 1). If the execution of T on P is correct or passed, then a mutation engine makes syntactic changes to the program P: the rule that specifies syntactic variations is defined as a* ***mutation operator****, and the result of one application of a mutation operator is a set of $mutants$ $\mathbb{M}$ (Step 2.2). Otherwise, fix P and go back to Step 1 (Step 2.1). After that, each mutant $P_m \in \mathbb{M}$ is executed against T to verify whether test cases in T fail or not (Step 3). The failure of the test execution means the mutant is killed. If all the mutants in $\mathbb{M}$ are dead or killed, then the process of mutation testing terminates (Step 4.1). Otherwise, check whether all live or surviving mutants are equivalent. If all live mutants are equivalent, then mutation testing is completed (Step 5.1). If not, improve T and go back to Step 1 until satisfying the condition(s) that all the mutants are dead and/or all live mutants are equivalent.*

The most widely-known *mutation operators* (also called mutant operators or mutation rules) include Arithmetic Operator Replacement (AOR), Relational Operator Replacement (ROR) and Absolute Value Insertion (ABS). Here is an example of AOR mutation operator: `X=a+b` is mutated to `X=a-b`, `X=a×b`, and `X=a÷b`.

The execution results of T on $P_m \in \mathbb{M}$ are compared with P: (1) if the output of P_m is different from P, then P_m is *killed* by T; (2) otherwise, i.e., the output of P_m is the same as P, this leads to either (2.1) P_m being *equivalent* to P, which means that they are syntactically different but functionally equivalent; or (2.2) T being not adequate to detect the mutants, which requires test case augmentation. The result of mutation testing can be represented by the *mutation score* (also referred to as mutation coverage or mutation adequacy), which is defined as:

$$mutation\_score = \frac{\#\,killed\_mutants}{\#nonequivalent\_mutants} \tag{1.1}$$

A mutation testing system can be regarded as a language system [47] since the pro-

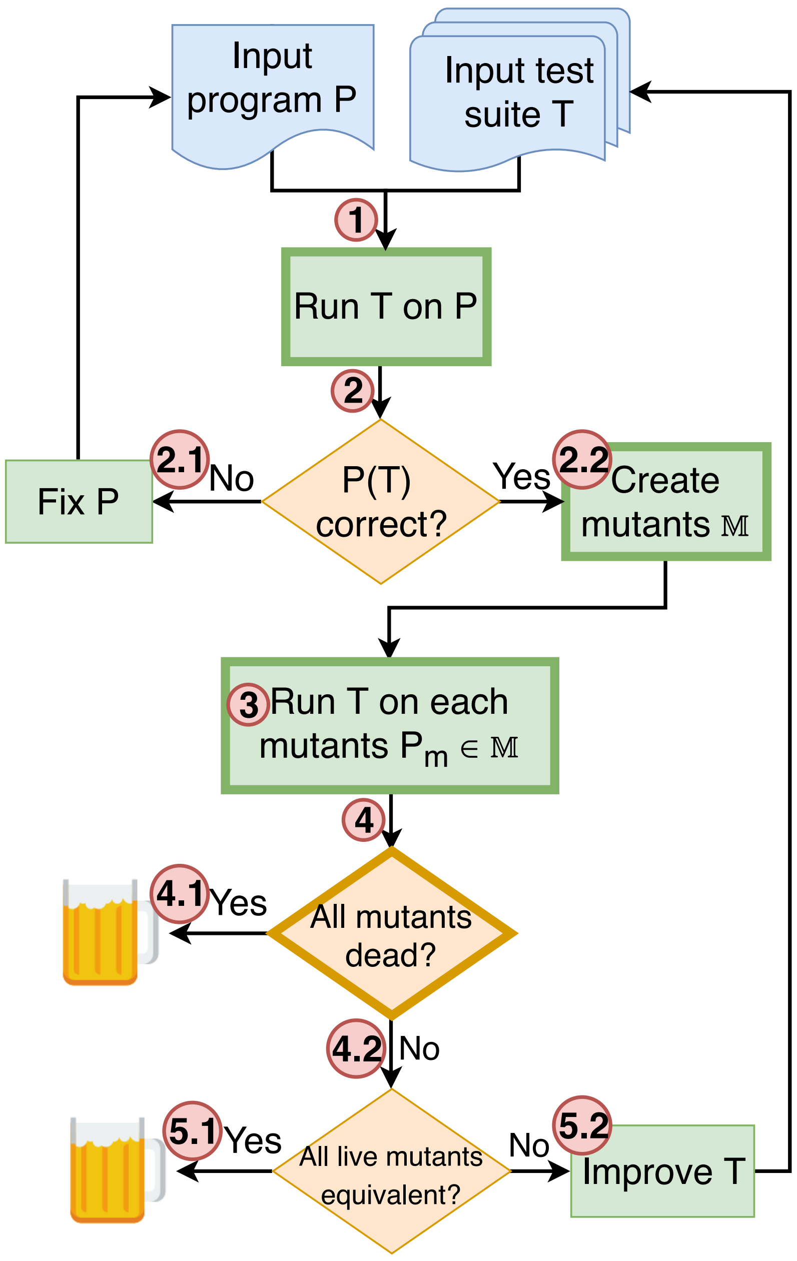


Figure 1.1: Generic process of mutation testing

grams under test must be parsed, modified and executed. The main components of mutation testing consist of the mutant creation engine, the equivalent mutant detector, and the test execution runner. There have been numerous mutation testing tools developed for different programming languages, such as Proteum [113] for C, Mujava [240] and PiTest [13] for Java, and SQLMutation [342] for SQL.

The *benefits* of mutation testing have been extensively investigated and can be summarised as [197]: 1) having better fault exposing capability compared to other test coverage criteria [144, 228, 244], 2) being an excellent alternative to real faults and providing a good indication of the fault detection ability of a test suite [50]. However, the *limita-*

tions of mutation testing are inherent. Firstly, it requires to re-run the test suites against each mutant, whose number increases exponentially with the size of the program under test [93]. Also, the *equivalent mutant detection* is an inevitable stage of mutation testing which is a prominent *undecidable problem* [91], thereby requiring human effort to investigate. In summary, the major limitations of mutation testing are as follows: 1) the high computational cost caused by generating and executing the numerous mutants and, 2) the tremendous time-consuming human investigation required for equivalent mutant detection.

To address the issue of high computational cost, several methods have been proposed and these can be classified in three main categories [277]: (*do fewer*) selecting fewer mutants to evaluate [41, 271], (*do smarter*) using run-time information to avoid unnecessary test executions [185, 200], (*do faster*) reducing the execution time for each single mutant [344]. As for the equivalent mutant problem, in Madeyski et al. [241]'s survey, they classified three main research directions: (1) detecting equivalent mutants, such as Baldwin and Sayward [63] (using compiler optimisations), and Martin and Xie [242] (through change-impact analysis); (2) avoiding equivalent mutant generation, such as Mresa and Bottaci [258] (through selective mutation), and Harman et al. [174] (using program dependence analysis); (3) suggesting equivalent mutants, such as dynamic invariants analysis [312], and coverage change examination (e.g., [314]).

More recently, the *usefulness of mutants* [203] has resulted in an increasing interest by researchers. Several studies claimed that the majority of the mutants generated by existing mutation operators are equivalent, trivial and redundant [88, 198, 203, 219, 295], which reduces the efficacy of the mutation score. To address this knowledge gap, numerous studies have investigated how useful mutants are. Example studies include *mutant subsumption* [219], *stubborn* mutants [373], and *real-fault coupling* [201, 295]. *Mutant subsumption*, which was proposed by Bob Kurtz et al. [219], is a graph model to describe the relationships among mutants, and can thus be used to eliminate the redundant mutants. *Stubborn mutants* are a set of mutants that remain alive and that cannot be shown to be equivalent [177]. Yao et al. [373] reported on the causes and prevalence of equivalent mutants and their relationship to stubborn mutants based on a manual analysis of 1230 mutants. The investigations on the relation between mutants and real faults (*real-fault coupling*) can be dated back to 1992 [118]. Recent works are based on large-scale empirical studies using real-world programs, such as Just et al. [201] and Papadakis et al. [295].

1.2. RESEARCH QUESTIONS

In this section, we summarise the research questions we propose for this thesis in order to achieve our goal: to investigate, improve and extend the applicability of mutation testing.

The first step is to investigate the current state of how mutation testing is applied. To understand *how mutation testing is actually applied* (**RQ2**[1] in Chapter 2), we start with a systematic literature review to summarise existing evidence concerning the main

[1]We number our research questions according to the chapter number of this thesis, thereby our research question starting from 2.

applications of mutation testing and identify limitations and gaps in current research. This is important because the existing surveys (e.g., [197, 241, 279]) only provide us with an overview of the most influential realisations of mutation testing in *research*, thereby omitting the perspective of how mutation testing is applied and what limitations are encountered in *research*. To achieve our goal, we first would like to identify and classify the main quality assurance processes where mutation testing is applied. This leads to **RQ2.1**:

> **RQ2.1** *How is mutation testing used in quality assurance processes*[2] *?*

During the synthesis of the empirical evidence related to mutation testing, we also became interested in how the related empirical studies are reported, so that we can provide guidelines for applying and reporting on mutation testing and recommendations for future work. **RQ2.2** addresses exactly this concern:

> **RQ2.2** *How are empirical studies related to mutation testing designed and reported?*

Based on a collection of 191 papers published between 1981 and 2015, our systematic literature review presents a detailed analysis of the application perspective of mutation testing, where we only selected papers that use mutation testing as a tool for evaluating or improving other quality assurance processes rather than focusing on the development of mutation tools, operators or challenges and open issues for mutation testing. Among the results, three further points attract our attention:

1. The high computational cost issue of mutation testing is not well-solved in the context of our research body.
2. A deeper understanding of mutation testing is required, such as what particular kinds of faults mutation testing is good at finding and what makes it hard to do mutation testing.
3. Most studies use mutation testing as an assessment tool targeting unit tests; as such, we feel that the application domain is quite limited.

The first item points us to the issue of the high computational cost when applying mutation testing. We investigate *whether we can use compression techniques to speed up mutation testing* (**RQ3** in Chapter 3). More specifically, we devise six compression techniques based on two clustering algorithms and three mutant selection strategies. The clustering algorithms we adopt to cluster mutants are *overlapped grouping* and *Formal Concept Analysis (FCA)*. The *overlapped* method is the simplest and strictest clustering method, i.e., elements are only grouped together if they are identical. Formal Concept Analysis (FCA) was originally a data analysis method and has shown to be a powerful mathematical technique to convey and summarise large amounts of information [364].

[2] The quality assurance processes include testing activities [71] and debugging [74] in general. In more specific, the quality assurance processes include all the daily work responsibilities of test engineers (e.g. designing test inputs, producing test case values, running test scripts, analysing results, and reporting results to developers and managers) [47].

FCA produces the *concept lattice* or *concept hierarchy* from a collection of objects and their properties. In the mutation testing context, our *FCA-based compression* technique only considers the maximal concepts that are directly connected to the exit point in the lattice hierarchy, the so-called *maximal groupings*.

To select mutants for execution, we leverage the knowledge of mutation locations and mutation operator types. To steer our investigation, we propose the following three research questions:

RQ3.1 *How accurate are different compression techniques?*
RQ3.2 *How do compression techniques perform in terms of speed-up?*
RQ3.3 *What is the trade-off between accuracy and speed-up for the compression techniques?*

The second point from our literature review addresses the necessity to deepen the understanding of mutation testing. The current research considers the mutation score mostly related the test suite quality and mutant usefulness [201, 219, 295, 373]. However, *how can production code quality in terms of testability and observability influence the mutation score?* (**RQ4** in Chapter 4) has rarely been explored. To answer this question, we investigate the relationship between testability and observability metrics and mutation testing. More specifically, we conjecture that *software testability* [89, 191] and *code observability* [80, 328] are two key factors. The following three research questions steer our investigation into the relationship between *testability, observability* and the mutation score:

RQ4.1 *What is the relation between testability metrics and the mutation score?*
RQ4.2 *What is the relation between observability metrics and the mutation score?*
RQ4.3 *What is the relation between the combination of testability and observability metrics and the mutation score?*

After investigating the relationship between *testability, observability* and the mutation score, we still lack insight into how these relationships can be made actionable for software engineers in practice. That is why, based on the observations from **RQ4.1**-**RQ4.3**, we define anti-patterns or indicators that software engineers can apply to their code to ensure that mutants can be killed. This leads us to **RQ4.4**:

RQ4.4 *To what extent does the refactoring of anti-patterns based on testability and observability help in improving the mutation score?*

The third message from our literature review indicates that the application domain of mutation testing is limited; this inspires us to explore more diverse applications for mutation testing: the one is mutation testing for *physical computing*, and the other is applying mutation testing to *GPU programs*.

For the first new application domain we have explored *physical computing* which builds interactive systems between the physical world and computers [282]. Physical computing, has been widely used in a wide variety of domains and applications, e.g., the

Internet of Things (IoT), wherein sensors and actuators blend seamlessly with the environment around us, and the information is shared across platforms [166]. Examples of physical computing applications (or cyber-physical systems) include medical devices and systems, aerospace systems, transportation vehicles and intelligent highways, defense systems, robotic systems, process control, factory automation, building and environmental control and smart spaces [302]. Compared to conventional software projects, the costs associated with failing physical computing systems are often even bigger, as bugs can result in real-life accidents [186]. For example, a robotic arm might accidentally hurt the human if the programmer does not set up the initial state properly. Therefore, to develop a rigorous and sound physical computing system, a high-quality test suite becomes crucial. This brings us to *mutation testing*, which has been shown to perform well in exposing faults [144, 228, 244].

To investigate *whether physical computing can benefit from mutation testing* (**RQ5** in Chapter 5), we propose a novel mutation testing approach for physical computing systems. Specifically, we define a set of mutation operators based on *common mistakes* that we have observed when developing physical computing systems. To assess the efficacy of our mutation testing approach, we implement a mutation testing tool for physical computing systems (coined MUTPHY) to answer the following questions:

RQ5.1 *How effective is* MUTPHY *in evaluating the existing test suite?*
RQ5.2 *How efficient is* MUTPHY *in generating non-equivalent mutants?*
RQ5.3 *Is it possible to kill all non-equivalent surviving mutants by adding extra test cases?*

GPU computing, the other application domain we have targeted, aims to use a GPU as a co-processor to accelerate CPUs for general-purpose scientific and engineering computing [283]. Compared to the CPU, the GPU contains many more transistors devoted to data processing rather than data caching and flow control [267]. Thus, the GPU is especially well-suited for compute-intensive, highly parallel computation.

Thanks to rapid advances in programmability and performance, we have observed that GPUs have been widely applied in High-Performance Computing (HPC) [134, 336], as well as safety-critical domains (e.g., medical science [332]). This attracts increasing attention in terms of *quality assurance* for GPU applications [158, 303]. Again, we come up with mutation testing, to investigate *if mutation testing can help in GPU program testing* (**RQ6** in Chapter 6). To enable mutation testing for GPU applications, we develop a mutation testing tool named MUTGPU. We steer our investigation along the four research questions:

RQ6.1 *How frequently can GPU-specific mutation operators be applied?*
RQ6.2 *How effective are conventional mutation operators in evaluating the test suite of GPU programs?*
RQ6.3 *How effective are GPU-specific mutation operators in evaluating the test suite of GPU programs?*
RQ6.4 *How do GPU-specific mutation operators compare with conventional mutation operators in terms of the improvement?*

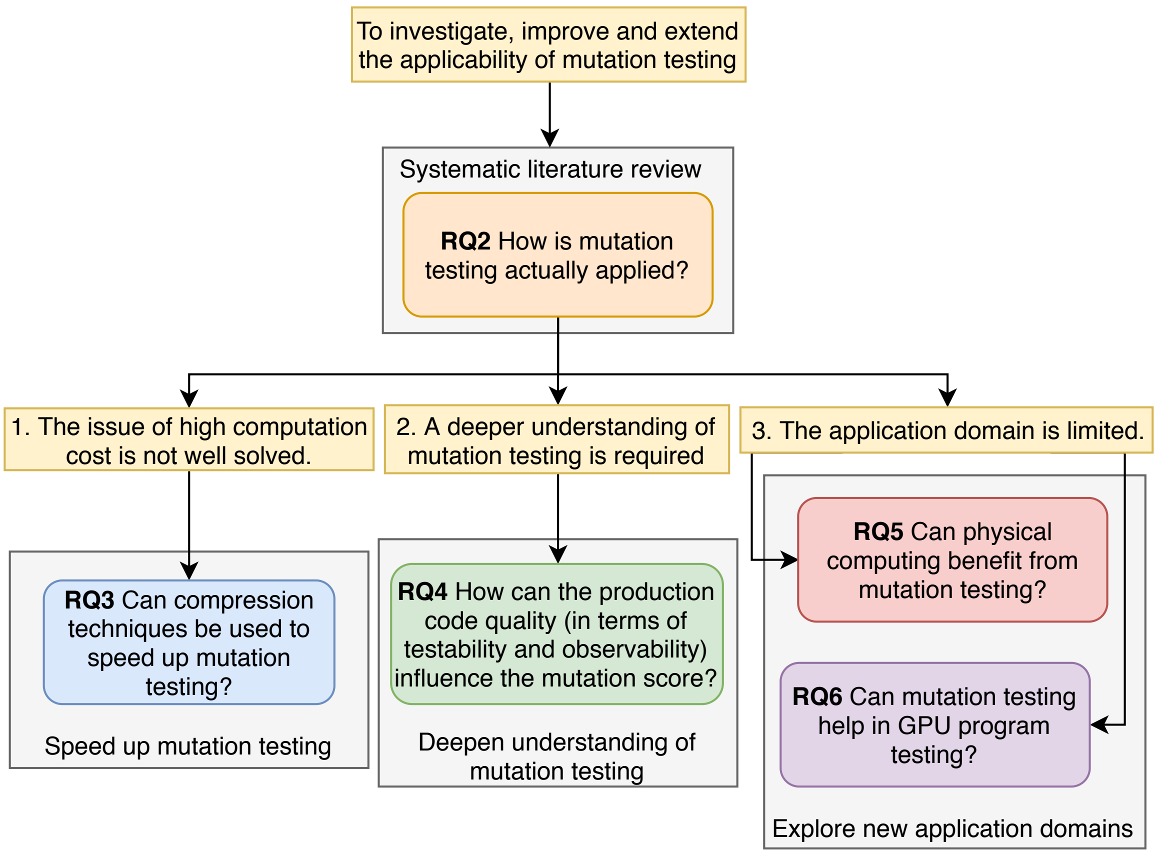

To investigate, improve and extend the applicability of mutation testing
Systematic literature review
RQ2 How is mutation testing actually applied?
1. The issue of high computation cost is not well solved.
2. A deeper understanding of mutation testing is required
3. The application domain is limited.
RQ3 Can compression techniques be used to speed up mutation testing?
Speed up mutation testing
RQ4 How can the production code quality (in terms of testability and observability) influence the mutation score?
Deepen understanding of mutation testing
RQ5 Can physical computing benefit from mutation testing?
RQ6 Can mutation testing help in GPU program testing?
Explore new application domains

Figure 1.2: Summary of research questions

To sum up, we present an overview of our research questions in Figure 1.2. As shown in Figure 1.2, we start with **RQ2** which seeks to find out how mutation testing is actually applied. We answer **RQ2** by conducting a systematic literature review, resulting in three interesting findings for further investigation. The first point “The issue of high computation cost is not well solved" leads us to **RQ3**. **RQ4** is generated from the second finding which requires us to deepen the understanding of mutation testing. More specifically, we investigate the relationship between production code quality in terms of testability/observability and mutation testing. The last message indicating that the current application domain is limited inspires us to explore new applications for mutation testing **RQ5** and **RQ6**.

1.3. Research Methodology

The main goal of this thesis is to explore ways to improve and extend the applicability of mutation testing. Therefore, our first step is to synthesise existing studies to gain insights into the state-of-the-art research regarding the application perspective of mutation testing. As *systematic literature reviews* [212] have been shown to be good tools to summarise existing evidence concerning a technology and identify gaps in current research [222], we follow this approach for reaching our objectives of **RQ2**.

The results from our systematic literature inspire and lead us to three directions of research: (1) speeding up mutation testing via compression techniques (**RQ3**); (2) deepening understanding of mutation testing by investigation of the relationship between testability, observability and the mutation score (**RQ4**); and (3) exploring new application domains of mutation testing (**RQ5** and **RQ6**). To answer **RQ3** to **RQ6**, we conduct a series of empirical studies by employing a combination of *quantitive* and *qualitative* approaches. Because quantitative strategies such as experimental studies are appropriate when testing the effect of some manipulation or activity, while a qualitative study of beliefs and understandings are appropriate to find out *why* the results from a quantitative investigation as they are [105]. We adopt different empirical methods to evaluate our proposed approaches or hypotheses, e.g., *experiment* (e.g., Chapter 3) and *case study* (e.g., Chapter 4).

In the following section, we would like to present an overview of the research methodology to show that we use different research methods in this thesis. For each research question, we are going to describe our research methods in more detail.

Chapter 2 For **RQ2**, we perform a *systematic literature review* [212] to investigate how mutation testing is actually applied. Unlike the general literature review, the systematic literature review aims to eliminate bias and incompleteness through a systematic mechanism [222]. We follow Kitchenham [212]'s guidelines for applying the systematic literature review specifically in the field of software engineering. To answer **RQ2.1** and **RQ2.2**, we generate an *attribute framework* to characterise each paper. The attribute framework consists of seven facets of interest that are highly relevant to the information we need to answer the questions. In this way, we can also show quantitative results for each attribute to support our answers.

Chapter 3 The goal of **RQ3** is to verify whether we can use compression techniques to speed up mutation testing. To achieve that, we conduct an *experimental study* with 20 open-source Java projects and using the test suites manually written by the original developers. To answer **RQ3.1** to **RQ3.3**, we evaluate six compression techniques together with two baselines: mutation sampling and weak mutation. This study mainly adopts *quantitive* research approaches by comparing the performance of different techniques in terms of accuracy and speed-up. In addition to a set of quantitive evaluation metrics, such as the absolute error and the overhead, we also adopt Friedman's test [304] with $\alpha = 0.05$ to assess whether the differences among the eight strategies are statistically significant or not. Also, in case we observe exceptions to the major findings, we also conduct a *qualitative* analysis to understand why.

Chapter 4 To answer **RQ4.1** to **RQ4.3**, we aim to investigate the relationship between testability/observability metrics and mutation testing. Thereby, we conduct an *experimental study* using six systems publicly available on GitHub. In this experiment, we first adopt Spearman's rank-order correlation to statistically measure the correlation between each metric (both existing code metrics and our newly proposed mutant observability metrics) and the mutation score of the corresponding methods or classes. Moreover, except for the pair-wise correlations between each metric and the mutation score, we are also interested in how those metrics interact with each other. To do so, we adopt

Random Forest to predict the mutation score based on those metrics. For **RQ4.4**, we perform a *case study* involving 16 code fragments to investigate whether the refactoring of anti-patterns improves the mutation score.

Chapter 5 To investigate **RQ5**, we first need a prototype tool to enable mutation testing for physical computing systems. So, we develop a mutation testing tool coined MUTPHY working on Raspberry Pi and Arduino platforms. To assess the efficacy of our mutation testing approach, we conducted an *experimental study*. To answer **RQ5.1** and **RQ5.2**, we compare the mutation score and the number of equivalent mutants quantitively. As for **RQ5.3**, we analyse the non-equivalent surviving mutants in detail and try to engineer new test cases to kill these mutants manually.

Chapter 6 Similar to **RQ5**, the first step to answer **RQ6** is to implement a tool to apply mutation testing in GPU programs, and then we conduct an *experimental study* using our self-implemented tool named MUTGPU. To answer **RQ6.1**, we evaluate the frequency of each mutation operator based on the number of generated mutants and the mutation score. For **RQ6.2** and **RQ6.3**, we determine the effectiveness of the mutation operators in assessing test quality of GPU programs based on non-equivalent surviving mutants, where we use both quantitative (by comparing the mutation scores) and qualitative (by manually analysing non-equivalent mutants) methods. To compare the conventional mutation operators with GPU-specific ones (**RQ6.4**), we first engineer new test cases to obtain a C-sufficient test suite for each system. Then, we manually analyse the remaining GPU mutants that cannot be detected by the C-sufficient test suites.

1.4. CONTRIBUTIONS AND THESIS OUTLINE

In this section, we are going to summarise the contributions of this thesis, and then present the outline of the thesis.

An in-depth literature survey on the *application perspective* of mutation testing (Chapter 2). We conduct a systematic literature review of 191 studies that apply mutation testing in scientific experiments, clearly contrasting previous literature reviews that surveyed the main development of mutation testing, and that did not specifically go into how mutation testing is applied (e.g., [197, 241, 279]).

A detailed attribute framework that generalises and details the essential elements related to the *actual* use of mutation testing (Chapter 2). Through a detailed reading of this research body, we derive an attribute framework that is consequently used to characterise the studies in a structured manner. The resulting attribute framework can be used as a reference for researchers when describing how they apply mutation testing.

A series of recommendations for future work including valuable suggestions on how to report mutation testing in testing experiments in an appropriate manner (Chapter 2). Based on the results of the literature review, we provide (1) guidelines on how to apply and subsequently report on mutation testing in testing experiments and (2) recommendations for future work.

A detailed investigation of different compression techniques to speed up mutation testing (Chapter 3). We propose six mutation compression strategies by leveraging mutant clustering algorithms and weak mutation. To evaluate the efficacy of our methods, we conduct a study on 20 open source projects. The overall results indicate that mutation compression techniques are a better choice than random sampling and weak mutation in practice: they can effectively speed up strong mutation 6.3 to 94.3 times with an accuracy of >90%.

A detailed investigation of the relationship between testability and observability metrics and the mutation score (Chapter 4). We collect 64 existing source code quality metrics for testability, and propose a set of metrics that specifically target mutant observability. Then, we use statistical analysis on open-source Java projects to investigate the relationship between testability, observability, and the mutation score. Last, we perform a case study involving 16 code fragments to investigate whether the refactoring of anti-patterns based on the aforementioned metrics can improve the mutation score.

A generic mutation testing approach for physical computing systems (Chapter 5). We propose a novel mutation testing approach for physical computing systems. Specifically, we define a set of mutation operators based on common mistakes that we observe when developing physical computing systems. Also, we implement a mutation testing tool coined MUTPHY for physical computing systems. We present an initial evaluation of our approach on the Raspberry Pi and Arduino platforms, which shows promising results.

A generic mutation testing approach for GPU programs (Chapter 6). First, we design nine new GPU-specific mutation operators in addition to conventional mutation operators. We develop a mutation testing tool named MUTGPU especially for GPU applications in the CUDA programming model. We perform an empirical study involving six GPU projects.

The remainder of the thesis is organised as follows: Chapter 2 presents a systematic literature review on the application of mutation testing. Chapter 3 investigates six mutation compression techniques to speed up mutation testing. Chapter 4 describes a detailed investigation of the relationship between testability and observability metrics and the mutation score. Chapter 5 introduces a novel mutation testing approach for physical computing systems, while Chapter 6 provides a study on applying mutation testing in GPU programming. Chapter 7 concludes the thesis.

2 A Systematic Literature Review of How Mutation Testing Supports Quality Assurance Processes

Mutation testing has been very actively investigated by researchers since the 1970s and remarkable advances have been achieved in its concepts, theory, technology and empirical evidence. While the most influential realisations have been summarised by existing literature reviews, we lack insight into how mutation testing is actually *applied. Our goal is to identify and classify the main applications of mutation testing and analyse the level of replicability of empirical studies related to mutation testing. To this aim, this chapter provides a systematic literature review on the* application perspective *of mutation testing based on a collection of 191 papers published between 1981 and 2015. In particular, we analysed in which quality assurance processes mutation testing is used, which mutation tools and which mutation operators are employed. Additionally, we also investigated how the inherent core problems of mutation testing, i.e., the equivalent mutant problem and the high computational cost, are addressed during the actual usage. The results show that most studies use mutation testing as an assessment tool targeting unit tests, and many of the supporting techniques for making mutation testing applicable in practice are still underdeveloped. Based on our observations, we made nine recommendations for future work, including an important suggestion on how to report mutation testing in testing experiments in an appropriate manner.*

This chapter has been published in Software Testing Verification Reliability (STVR) [393].

2.1. Introduction

Mutation testing is defined by Jia and Harman [197] as a fault-based testing technique which provides a testing criterion called the *mutation adequacy score*. This score can be used to measure the effectiveness of a test set in terms of its ability to detect faults [197]. The principle of mutation testing is to introduce syntactic changes into the original program to generate faulty versions (called *mutants*) according to well-defined rules (mutation operators) [279]. Mutation testing originated in the 1970s with works from Lipton [230], DeMillo et al. [117] and Hamlet [168] and has been a very active research field over the last few decades. The activeness of the field is in part evidenced by the extensive survey of more than 390 papers on mutation testing that Jia and Harman published in 2011 [197]. Jia and Harman's survey highlights the research achievements that have been made over the years, including the development of tools for a variety of languages and empirical studies performed [197]. Additionally, they highlight some of the actual and inherent problems of mutation testing, amongst others: (1) the high computational cost caused by generating and executing the numerous mutants and (2) the tremendous time-consuming human investigation required by the test oracle problem and equivalent mutant detection.

While existing surveys (e.g., [197, 241, 279]) provide us with a great overview of the most influential realisations in research, we lack insight into how mutation testing is actually *applied*. Specifically, we are interested in analysing in which quality assurance processes mutation testing is used, which mutation tools are employed and which mutation operators are used. Additionally, we want to investigate how the aforementioned problems of the high computational cost and the considerable human effort required are dealt with when applying mutation testing. In order to steer our research, we aim to fulfil the following objectives:

- to identify and classify the applications of mutation testing in quality assurance processes;
- to analyse how the main problems are coped with when applying mutation testing;
- to provide guidelines for applying mutation testing in testing experiments;
- to identify gaps in current research and to provide recommendations for future work.

As systematic literature reviews have been shown to be good tools to summarise existing evidence concerning a technology and identify gaps in current research [212], we follow this approach for reaching our objectives. We only consider the articles which provide sufficient details on how mutation testing is used in their studies, i.e., we require at least a brief specification about the adopted mutation tool, mutation operators or mutation score. Moreover, we selected only papers that use mutation testing as a tool for evaluating or improving other quality assurance processes rather than focusing on the development of mutation tools, operators or challenges and open issues for mutation testing. This resulted in a collection containing 191 papers published from 1981 to 2015. We analysed this collection in order to answer the following two research questions:

RQ2.1: *How is mutation testing used in quality assurance processes?*

This research question aims to identify and classify the main software testing tasks where mutation testing is applied. In particular, we are interested in the following key aspects: (1) in which circumstances mutation testing is used (e.g., assessment tool), (2) which quality assurance processes are involved (e.g., test data generation, test case prioritisation), (3) which test level it targets (e.g., unit level) and (4) which testing strategies it supports (e.g., structural testing). The above four detailed aspects are defined to characterise the essential features related to the usage of mutation testing and the quality assurance processes involved. With these elements in place, we can provide an in-depth analysis of the applications of mutation testing.

RQ2.2: *How are empirical studies related to mutation testing designed and reported?*

The objective of this question is to synthesise empirical evidence related to mutation testing. The case studies or experiments play an inevitable role in a research study. The design and demonstration of the evaluation methods should ensure the replicability. For replicability, we mean that the subject, the basic methodology, as well as the result, should be clearly pointed out in the article. In particular, we are interested in how the articles report the following information related to mutation testing: (1) mutation tools, (2) mutation operators, (3) mutant equivalence problem, (4) techniques for reduction of computational cost and (5) subject programs used in the case studies. After gathering this information, we can draw conclusions from the distribution of related techniques adopted under the above five facets and thereby provide guidelines for applying mutation testing and reporting the used setting/tools.

The remainder of this review is organised as follows: Section 2.2 provides an overview on background notions on mutation testing. Section 2.3 details the main procedures we followed to conduct the systematic literature review and describes our inclusion and exclusion criteria. Section 2.4 presents the discussion of our findings, particularly Section 2.4.3 summarises the answers to the research questions, while Section 2.4.4 provides recommendations for future research. Section 2.5 discusses the threats to validity, and Section 2.6 concludes the chapter.

2.2. BACKGROUND

In order to level the playing field, we first provide the basic concepts related to mutation testing, i.e., its fundamental hypothesis and generic process, including the *Competent Programmer Hypothesis*, the *Coupling Effect*, *mutation operators* and the *mutation score*. Subsequently, we discuss the benefits and limitations of mutation testing. After that, we present a historical overview of mutation testing where we mainly address the studies that concern the application of mutation testing.

2.2.1. BASIC CONCEPTS

FUNDAMENTAL HYPOTHESIS

Mutation testing starts with the assumption of the *Competent Programmer Hypothesis* (introduced by DeMillo et al. [117] in 1978): "*The competent programmers create programs that are close to being correct.*" This hypothesis implies that the potential faults in the programs delivered by the competent programmers are just very simple mistakes; these defects can be corrected by a few simple syntactical changes. Inspired by the above hypothesis, mutation testing typically applies small syntactical changes to original programs, thus implying that the faults that are seeded resemble faults made by "competent programmers".

At first glance, it seems that the programs with complex errors cannot be explicitly generated by mutation testing. However, the *Coupling Effect*, which was coined by DeMillo et al. [117] states that "*Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors*". This means complex faults are *coupled* to simple faults. This hypothesis was later supported by Offutt [268, 269] through empirical investigations over the domain of mutation testing. In his experiments, he used first-order mutants, which are created by applying the mutation operator to the original program once, to represent simple faults. Conversely, higher-order mutants, which are created by applying mutation operators to the original program more than once, stand for complex faults. The results showed that the test data generated for first-order mutants killed a higher percentage of mutants when applied to higher-order mutants, thus yielding positive empirical evidence about the *Coupling Effect*. Besides, there has been a considerable effort in validating the coupling effect hypothesis, amongst others the theoretical studies of Wah [358–360] and Kapoor [208].

THE GENERIC MUTATION TESTING PROCESS.

After introducing the fundamental hypotheses of mutation testing, we are going to give a detailed description of the *generic process* of mutation testing:

> *Given a program P and a test suite T, a mutation engine makes syntactic changes to the program P: the rule that specifies syntactic variations are defined as a $mutation\ operator$, and the result of one application of a mutation operator is a set of $mutants$ $\mathbb{M}$. After that, each mutant $P_m \in \mathbb{M}$ is executed against T to verify whether test cases in T fail or not.*

Here is an example of a mutation operator, i.e., Arithmetic Operator Replacement (AOR), on a statement `X=a+b`. The produced mutants include `X=a-b`, `X=a×b`, and `X=a÷b`.

The execution results of T on $P_m \in \mathbb{M}$ are compared with P: (1) if the output of P_m is different from P, then P_m is *killed* by T; (2) otherwise, i.e., the output of P_m is the same as P, this leads to either (2.1) P_m is *equivalent* to P, which means that they are syntactically different but functionally equivalent; or (2.2) T is not adequate to detect the mutants, which requires test case augmentation.

The result of mutation testing can be summarised using the *mutation score* (also referred to as mutation coverage or mutation adequacy), which is defined as:

$$mutation\ score = \frac{\#\ killed\ mutants}{\#\ nonequivalent\ mutants} \tag{2.1}$$

From the equation above, we can see that the detection of equivalent mutants is done before calculating the mutation score, as the denominator explicitly mentions non-equivalent mutants. Budd and Angluin [91] have theoretically proven that the equivalence of two programs is not decidable. Meanwhile, in their systematic literature survey, Madeyski et al. [241] have also indicated that the equivalent mutant problem takes an enormous amount of time in practice.

A mutation testing system can be regarded as a language system [47] since the programs under test must be parsed, modified and executed. The main components of mutation testing consist of the mutant creation engine, the equivalent mutant detector, and the test execution runner. The first prototype of a mutation testing system for Fortran was proposed by Budd and Sayward [90] in 1977. Since then, numerous mutation tools have been developed for different languages, such as Mothra [210] for Fortran, Proteum [113] for C, Mujava [240] for Java, and SQLMutation [342] for SQL.

Benefits & Limitations

Mutation testing is widely considered as a "high end" test criterion [47]. This is in part due to the fact that mutation testing is extremely hard to satisfy because of the massive number of mutants. However, many empirical studies found that it is much stronger than other test adequacy criteria in terms of fault exposing capability, e.g., Mathur and Wong [244], Frankl et al. [144] and Li et al. [228]. In addition to comparing mutation testing with other test criteria, there have also been empirical studies comparing real faults and mutants. The most well-known research work on such a topic is by Andrews et al. [50]: they suggest that when using carefully selected mutation operators and after removing equivalent mutants, mutants can provide a good indication of the fault detection ability of a test suite. As a result, we consider the benefits of mutation testing to be:

- better fault exposing capability compared to other test coverage criteria, e.g., all-use;
- a good alternative to real faults which can provide a good indication of the fault detection ability of a test suite.

The limitations of mutation testing are inherent. Firstly, both the generation and execution of a vast number of mutants are computationally expensive. Secondly, the equivalent mutant detection is also an inevitable stage of mutation testing which is a prominent undecidable problem, thereby requiring human effort to investigate. Thus, we consider the major limitations of mutation testing to be:

- the high computational cost caused by the large number of mutants;
- the undecidable Equivalent Mutant Problem resulting in the difficulty of fully automating the equivalent mutant analysis.

To deal with the two limitations above, a lot of research effort has been devoted to reduce the computational cost and to propose heuristics to detect equivalent mutants. As for the high computational cost, Offutt and Untch [277] performed a literature review in which they summarised the approaches to reduce computational cost into three strategies: *do fewer*, *do smarter* and *do faster*. These three types were later classified into two classes by Jia and Harman [197]: reduction of the generated mutants and reduction of the execution cost. Mutant sampling (e.g., [41, 367]), mutant clustering (e.g., [188, 195]) and selective mutation (e.g., [258, 271, 322]) are the most well-known techniques for reducing the number of mutants while maintaining efficacy of mutation testing to an acceptable degree. For reduction of the execution expense, researchers have paid much attention to weak mutation (e.g., [116, 185, 272]) and mutant schemata (e.g., [344, 345]).

To overcome the Equivalent Mutant Problem, there are mainly three categories classified by Madeyski et al. [241]: (1) detecting equivalent mutants, such as Baldwin and Sayward [63] (using compiler optimisations), Hierons et al. [177] (using program slicing), Martin and Xie [242] (through change-impact analysis), Ellims et al. [132] (using running profile), and du Bousquet and Delaunay [129] (using model checker); (2) avoiding equivalent mutant generation, such as Mresa and Bottaci [258] (through selective mutation), Harman et al. [174] (using program dependence analysis), and Adamopoulos et al. [42] (using co-evolutionary search algorithm); (3) suggesting equivalent mutants, such as bayesian learning [350], dynamic invariants analysis [312], and coverage change examination (e.g. [314]).

2.2.2. Historical Overview

In this subsection, we are going to present a chronological overview of important research in the area of mutation testing. As the focus of our review is the application perspective of mutation testing, we mainly address the studies that concern the application of mutation testing. In the following paragraphs, we will first give a brief summary of the development of mutation testing, and — due to the sheer size of the research body — we will then highlight some notable studies on applying mutation testing.

Mutation testing was initially introduced as a fault-based testing method which was regarded as significantly better at detecting errors than the *covering measure* approach [92]. Since then, mutation testing has been actively investigated and studied thereby resulting in remarkable advances in its concepts, theory, technology and empirical evidence. The main interests in the area of mutation testing include (1) defining mutation operators [43], (2) developing mutation testing systems[185, 210, 240], (3) reducing the cost of mutation testing [271, 345], (4) overcoming the equivalent mutant detection problem [241], and (5) empirical studies with mutation testing [50]. For more literature on mutation testing, we refer to the existing surveys of DeMillo [115], Offutt and Untch [277], Jia and Harman [197] and Offutt [279].

In the meanwhile, mutation testing has also been applied to support other testing activities, such as test data generation and test strategy evaluation. The early application of mutation testing can be traced back to the 1980s [130, 264–266]). Ntafos is one of the very first researchers to use mutation testing as a measure of test set effectiveness. Ntafos applied mutation operators (e.g., constant replacement) to the source code of 14 Fortran programs [265]. The generated test suites were based on three test strategies, i.e., random testing, branch testing and data-flow testing, and were evaluated regarding mutation score.

DeMillo and Offutt [116] are the first to automate test data generation guided by fault-based testing criteria. Their method is called Constraint-based testing (CBT). They transformed the conditions under which mutants will be killed (necessity and sufficiency condition) to the corresponding algebraic constraints (using constraint template table). The test data was then automatically generated by solving the constraint satisfaction problem using heuristics. Their proposed constraint-based test data generator is limited and was only validated on five laboratory-level Fortran programs. Other remarkable approaches of the automatic test data generation includes a paper by Zhang et al. [382], who adopted Dynamic Symbolic Execution, and a framework by Papadakis and Malevris [293] in which three techniques, i.e., Symbolic Execution, Concolic Testing and Search-based Testing, were used to support the automatic test data generation.

Apart from test data generation, mutation testing is widely adopted to assess the cost-effectiveness of different test strategies. The work above by Ntafos [265] is one of the early studies on applying mutation testing. Recently, there has been a considerable effort in the empirical investigation of structural coverage and fault-finding effectiveness, including Namin and Andrews [262] and Inozemtseva et al. [189]. Zhang and Mesbah [384] proposed assertion coverage, while Whalen et al. [362] presented observable modified condition/decision coverage (OMC/DC); these novel test criteria were also evaluated via mutation testing.

Test case prioritisation is one of the practical approaches to reducing the cost of regression testing by rescheduling test cases to expose the faults as earlier as possible. Mutation testing has also been applied to support test case prioritisation. Among these studies, two influential papers are Rothermel et al. [307] and Elbaum et al. [131] who proposed a new test case prioritisation method based on the rate of mutants killing. Moreover, Do and Rothermel [127, 128] measured the effectiveness of different test case prioritisation strategies via mutation faults since Andrews et al.'s empirical study suggested that mutation faults can be representative of real faults [50].

The test-suite reduction is another testing activity we identified which is supported by mutation testing. The research work of Offutt et al. [275] is the first to target test-suite reduction strategies, especially for mutation testing. They proposed *Ping-Pong* reduction heuristics to select test cases based on their mutation scores. Another notable work is Zhang et al. [381] that investigated test-suite reduction techniques on Java programs with real-world JUnit test suites via mutation testing.

Another portion of the application of mutation testing is debugging, such as fault localisation. Influential examples include an article by Zhang et al. [361] in which mutation testing is adopted to investigate the effect of coincidental correctness in the context of a coverage-based fault localisation technique, and a novel fault localisation method by Papadakis et al. [290], [291] who used mutants to identify the faulty program statements.

2.2.3. COMPARISONS WITH EXISTING LITERATURE SURVEYS

In this section, we summarise the existing literature surveys on mutation testing and compare these surveys to our literature review. Table 2.1 lists seven literature surveys which we have found so far, including the years which the survey covered, whether the survey is a systematic literature review and the survey's main idea.

First of all, the scope of our literature review is different from the existing literature surveys. The surveys of DeMillo [115], Woodward [368], Offutt and Untch [277], Offutt [279] and Jia and Harman [197] focused on the development of mutation testing, where they summarised and highlighted the most influential realisations and findings on mutation testing. In the insightful works of Offutt and Untch [277], Offutt [279] and Jia and Harman [197], they only mentioned some of the most crucial studies which

Table 2.1: Summary of existing surveys on mutation testing

Survey	**Covered years**	**SLR?**	**Main idea**
DeMillo [115]	1978-1989	No	Summarise the conceptual basis, development of the mutation testing at the early stage
Woodward [368]	1978-1989	No	Review the mutation testing techniques of strong, weak and firm mutation
Offutt and Untch [277]	1977-2000	No	Review the history of mutation testing and the existing optimisation techniques for mutation testing
Offutt [279]	1977-2010	No	Review past mutation analysis research starting with the Mothra project, and summarise new trends of applications of mutation testing
Jia and Harman [197]	1977-2009	No	Provide a comprehensive analysis and survey of Mutation Testing, including theories, problems, cost reduction techniques, applications, empirical evaluation, and tools
Madeyski et al. [241]	1979-2010	Yes	Present a systematic literature review in the field of the equivalent mutant problem
Hanh et al. [170]	1991-2014	No	Analyse and conduct a survey on generating test data based on mutation testing

Note: Column '**SLR?**' means whether the paper is a systematic literature review or not.

applied mutation testing to support quality assurance processes, thus, the relevant research questions posed by us could not be answered by their reviews. Madeyski et al. [241] reviewed the equivalent mutant problem which is a subarea of mutation testing. Compared to their survey work, we are more interested in how approaches for detecting equivalent mutant are actually *used* in a research context. Hanh et al. [170] analysed the literature on mutation-based test data generation, which is a subset of our literature review. Our literature review not only covers the test data generation but also other quality assurance processes, e.g., test case prioritisation and debugging.

Moreover, our literature review follows the systematic literature review (SLR) methodology [85] which is not the case for six other literature reviews (Madeyski et al. [241] being the exception): we aim to review the existing articles in a more systematic way and provide a more complete list of the existing works on how mutation testing is actually applied in quality assurance processes. It is important to mention, though, that taking a subset of Offutt and Untch [277], Offutt [279] and Jia and Harman [197]'s results regarding quality assurance applications will not give as complete a view on quality assurance applications as our SLR actually does.

2.3. Research Method

In this section, we describe the main procedures we took to conduct this review. We adopted the methodology of the systematic literature review. A systematic literature review [212] is a means of aggregating and evaluating all the related primary studies under a research scope in an unbiased, thorough and trustworthy way. Unlike the general literature review, the systematic literature review aims to eliminate bias and incompleteness through a systematic mechanism [222]. Kitchenham [212] presented comprehensive and reliable guidelines for applying the systematic literature review to the field of software engineering. The guidelines cover three main phases: (i) planning the review, (ii) conducting the review, and (iii) reporting the review. Each step is well-defined and well-structured. By following these guidelines, we can reduce the likelihood of gener-

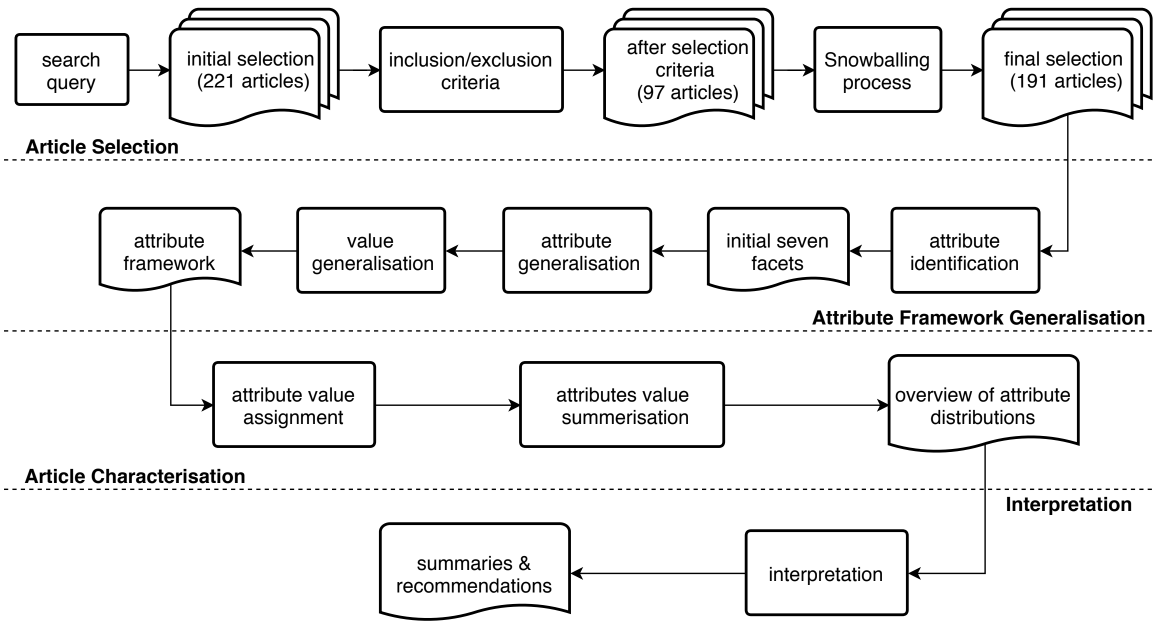

search query
initial selection (221 articles)
inclusion/exclusion criteria
after selection criteria (97 articles)
Snowballing process
final selection (191 articles)
Article Selection
attribute framework
value generalisation
attribute generalisation
initial seven facets
attribute identification
Attribute Framework Generalisation
attribute value assignment
attributes value summerisation
overview of attribute distributions
Article Characterisation
Interpretation
summaries & recommendations
interpretation

Figure 2.1: Overview of the systematic review process [106]

ating biased conclusions and sum all the existing evidence in a manner that is fair and seen to be fair.

The principle of the systematic literature review [85] is to convert the information collection into a systematic research study; this research study first defines several specific research questions and then searches for the best answers accordingly. These research questions and search mechanisms (consisting of study selection criteria and data extraction strategy) are included in a review protocol, a detailed plan to perform the systematic review. After developing the review protocol, the researchers need to validate this protocol for further resolving the potential ambiguity.

Following the main stages of the systematic review, we will introduce our review procedure in four parts: we will first specify the research questions, and then present the study selection strategy and data extraction framework. In the fourth step, we will show the validation results of the review protocol. The overview of our systematic review process is shown in Figure 2.1.

2.3.1. RESEARCH QUESTIONS

The research questions are the most critical part of the review protocol. The research questions determine study selection strategy and data extraction strategy. In this review, our objective is to examine the primary applications of mutation testing and identify limitations and gaps. Therefore, we can provide guidelines for applying mutation testing and recommendations for future work. To achieve these goals and starting with our most vital interests, the application perspective of mutation testing, we naturally further divide it into two aspects: (1) how mutation testing is used and (2) how the related empirical studies are reported. For the first aspect, we aim to identify and classify the main applications of mutation testing:

RQ2.2.1: **How is mutation testing used in quality assurance processes [1]?**

To understand how mutation testing is used, we should first determine in which circumstances it is used. The usages might range from using mutation testing as a way to assess how other testing approaches perform or mutation testing might be a building block of an approach altogether. This leads to RQ2.1.1:

RQ2.1.1: *Which role does mutation testing play in quality assurance processes?*

There is a broad range of quality assurance processes that can benefit from the application of mutation testing, e.g., fault localisation and test data generation. RQ1.2 seeks to uncover these activities.

RQ2.1.2: *Which quality assurance process does mutation testing support?*

In Jia and Harman's survey [197] of mutation testing, they found that most approaches work at the unit testing level. In RQ2.1.3 we investigate whether the application of mutation testing is also mostly done at the unit testing level, or whether other levels of testing have been also investigated in the literature.

RQ2.1.3: *Which test level does mutation testing target?*

Jia and Harman [197] have also indicated that mutation testing is most often used in a white-box testing context. In RQ2.1..4 we explore what other testing strategies can also benefit from the application of mutation testing.

RQ2.1.4: *Which testing strategies does mutation testing support?*

For the second aspect, we are going to synthesise empirical evidence related to mutation testing:

RQ2.2: **How are empirical studies related to mutation testing designed and reported?**

A plethora of mutation testing tools exist and have been surveyed by Jia and Harman [197]. Little is known which ones are most applied and why these are more popular. RQ2.2.1 tries to fill this knowledge gap by providing insight into which tools are used more frequently in a particular context.

RQ2.2.1: *Which mutation testing tools are being used?*

The mutation tools that we surveyed implement different mutation operators. Also, the various mutation approaches give different names to virtually the same mutation operators. RQ2.2.2 explores what mutation operators each method or tool has to offer and how mutation operators can be compared.

[1]The quality assurance processes include testing activities and debugging in general. In more specific, the quality assurance processes include all the daily work responsibilities of test engineers (e.g. designing test inputs, producing test case values, running test scripts, analyzing results, and reporting results to developers and managers) [47].

> **RQ2.2.2**: *Which mutation operators are being used?*

The equivalent mutant problem, i.e., the situation where a mutant leads to a change that is not observable in behaviour, is one of the most significant open issues in mutation testing. Both Jia and Harman [197] and Madeyski et al.[241] highlighted some of the most remarkable achievements in the area, but we have a lack of knowledge when it comes to how the equivalent mutant problem is coped with in practice when applying mutation testing during quality assurance activities. RQ2.2.3 aims to seek answers for exactly this question.

> **RQ2.2.3**: *Which approaches are used to overcome the equivalent mutant problem when applying mutation testing?*

As mutation testing is computationally expensive, techniques to reduce costs are important. Selective Mutation and Weak Mutation are the most widely studied cost reduction techniques [197], but it is unclear which reduction techniques are actually used when applying mutation testing, which is the exact topic of RQ2.2.4.

> **RQ2.2.4**: *Which techniques are used to reduce the computational cost when applying mutation testing?*

To better understand in which context mutation testing is applied, we want to look into the programming languages that have been used in the experiments. However, also the size of the case study systems is of interest, as it can be an indication of the maturity of certain tools. Finally, we are also explicitly looking at whether the case study systems are available for replication purposes (in addition to the check for availability of the mutation testing tool in RQ2.2.1).

> **RQ2.2.5**: *What subjects are being used in the experiments (regarding programming language, size, and data availability)?*

2.3.2. STUDY SELECTION STRATEGY

Initial Study Selection. We started with search queries in online platforms, including Google Scholar, Scopus, ACM Portal, IEEE Explore as well as Springer, Wiley, Elsevier Online libraries, to collect papers containing the keywords “mutation testing" or “mutation analysis" in their titles, abstracts, and keywords. Meanwhile, to ensure the high quality of the selected papers, we considered the articles published in seven top journals and ten top conferences (as listed in Table 2.2) dating from 1971 as data sources. The above 17 venues are chosen because they report a high proportion of research on software testing, debugging, software quality and validation. Moreover, we excluded article summaries, interviews, reviews, workshops[2], panels and poster sessions from the search. If the paper’s language is not English, we also excluded such a paper. After this step, 220 papers were initially selected.

Inclusion/Exclusion Criteria. Since we are interested in how mutation testing is *applied* in a research context, *thereby not excluding industrial practice*, we need selection

[2]In the snowballing procedure, we took the “Mutation Testing” workshop series into consideration, since this is the closest venue to mutation testing.

Table 2.2: Venues involved in study selection

Type	Venue Name	No. of papers After Applying Search queries	No. of papers After Applying In./Ex. Criteria	No. of papers After Snow-balling procedure
Journal	Journal of Empirical Software Engineering (EMSE)	4	3	6
	Information and Software Technology (IST)	0	0	3
	Journal Software Maintenance and Evolution (JSME)	0	0	0
	Software Quality Journal (JSQ)	0	0	2
	Journal of Systems and Software (JSS)	17	8	9
	Journal on Software Testing, Verification and Reliability (STVR)	33	16	23
	Transaction on Software Engineering and Methodology (TOSEM)	3	2	4
	Transaction on Reliability (TR)	1	1	1
	Transaction on Software Engineering (TSE)	19	9	21
Conference	**Proceedings Asia Pacific Software Engineering Conference (APSEC)**	0	0	1
	International Conference on Automated Software Engineering (ASE)	7	3	7
	European Software Engineering Conference / International Symposium on the Foundations of Software Engineering (ESEC/FSE)	6	1	9
	International Symposium on Empirical Software Engineering and Measurement (ESEM/ISESE)	2	1	3
	International Conference on Software Engineering (ICSE)	29	9	22
	International Conference on Software Maintenance and Evolution (ICSME/ICSM)	6	3	9
	International Conference on Software Testing, Verification, Validation (ICST)	45	23	22
	International Symposium on Software Reliability Engineering (ISSRE)	26	10	20
	International Symposium on Software Testing and Analysis (ISSTA)	14	3	12
	Proceedings International Conference on Quality Software (QSIC)	8	5	6
	Proceedings International Symposium on Search Based Software Engineering (SSBSE)	0	0	1
	Proceedings of the International Conference on Testing Computer Software (TCS)	0	0	1
Workshop	**International Workshop on Mutation Analysis**	0	0	9
	Total	220	97	191

Note: the venues marked in **bold** font are not initially selected, but where added after the snowballing procedure. We listed the venues alphabetically according to their abbreviations, e.g., EMSE is ahead of IST.

criteria to include the papers that use mutation testing as a tool for evaluating or improving other quality assurance processes and exclude the papers focusing on the development of mutation tools and operators, or challenges and open issues for mutation testing. Moreover, the selected articles should also provide sufficient evidence for answering the research questions. Therefore, we define two inclusion/exclusion criteria for study selection. The inclusion/exclusion criteria are as follows:

1. The article must focus on the supporting role of mutation testing in the quality assurance processes. This criterion excludes the research *solely* on mutation testing itself, such as defining mutation operators, developing mutation systems, investigating ways to solve open issues related to mutation testing and comparisons between mutation testing and other testing techniques.

2. The article exhibits sufficient evidence that mutation testing is used to support testing related activities. The sufficient evidence means that the article must clearly describe how the mutation testing is involved in the quality assurance processes. The author(s) must state at least one of the following details about the mutation

testing in the article: mutation tool, mutation operators, mutation score[3]. This criterion also excludes theoretical studies on mutation testing.

The first author of this SLR then carefully read the titles and abstracts to check whether the papers in the initial collection belong to our set of selected papers based on the inclusion/exclusion criteria. If it is unclear from the titles and abstracts whether mutation testing was applied, the entire article especially the experiment part was read as well. After we have applied the inclusion/exclusion criteria, 97 papers remained.

Snowballing Procedure. After selecting 97 papers from digital databases and applying our selection criteria, there is still a high potential to miss articles of interest. As Brereton et al. [85] pointed out, most online platforms do not provide adequate support for systematic identification of relevant papers. To overcome this shortfall of online databases, we then adopted both backward and forward snowballing strategies [365] to find missing papers. Snowballing refers to using the list of references in a paper or the citations to the paper to identify additional papers [365]. Using the references and the citations respectively are referred to as backward and forward snowballing [365].

We used the 97 papers as the starter set and performed a backward and forward snowballing procedure recursively until no further papers could be added to our set. During the snowballing procedure, we extended the initially selected venues to minimise the chance of missing related papers. The snowballing process resulted in another 82 articles (and five additional venues). The International Workshop on Mutation Analysis was added during the snowballing procedure.

To check the completeness of the initial study collection, we first ran a reference check based on Jia et al.'s survey (among 264 papers) as our literature review was initially motivated from their paper. The results showed that: (1) 5 papers have already been included in our collection; (2) 3 additional papers that should be included; and (3) 246 papers are excluded. Again, we applied snowballing techniques to the additional three papers, and the three papers resulted in a total of 12 papers for our final collection (191 papers in total[4]).

Furthermore, we ran a sanity check on our final collection to examine how many papers do not have the keywords "mutation analysis" or "mutation testing" in their abstracts, titles or keywords. The sanity check resulted in 112 papers; 15 papers are missing in the initial data collection by applying search queries in online platforms. Most of the missing papers (10 out of 15) (e.g., Offutt et al. [275] and Knauth et al.[213]) are not from the pre-considered 17 venues. The results of the sanity check indicate that there are potentials of missing papers based on search queries in online platforms; however, the snowballing procedure can minimise the risks of missing papers.

The detailed records of each step can be found in our GitHub repository [23].

2.3.3. Data Extraction Strategy

Data extracted from the papers are used to answer the research questions we formulated in Section 2.3.1. Based on our research questions, we draw seven facets of interest

[3]The studies which *merely* adopted hand-seeded faults which are not based on a set of mutation operators are not part of this survey.

[4]We did not control for double counting here as there are usually additional experiments and further discussion in the extended version.

that are highly relevant to the information we need to answer the questions. The seven facets are: (1) the roles of mutation testing in quality assurance processes; (2) the quality assurance processes; (3) the mutation tools used in experiments; (4) the mutation operators used in experiments; (5) the description of equivalent mutant problem; (6) the description of cost reduction techniques for mutation testing; (7) the subjects involved in experiments. An overview of these facets is given in Table 2.3.

For each facet, we first read the corresponding details in each paper and extracted the exact text from the papers. During the reading procedure, we started by identifying and classifying attributes of interest under each facet and assigned values to each attribute. The values of each attribute were generalised and modified during the reading process: we merged some values or divided one into several smaller groups. In this way, we generated an attribute framework, and then we used the framework to characterise each paper. Therefore, we can show quantitative results for each attribute to support our answers. Moreover, the attribute framework can also be further used for validation and replication of the review work. To categorise the attributes for each paper, all the abstracts, introductions, empirical studies and conclusions of the selected papers were carefully read. If these sections were not clear or were somehow confusing, we also took other sections from the paper into consideration. Furthermore, for categorising the test level attribute, to minimise misinterpretations of the original papers, we looked beyond the aforementioned sections to determine the test level (i.e., “unit", “module", “integration", “system" and “acceptance" [47]). In particular, we used the former five words as keywords to search for the entire paper. If this search yielded no results, we did not necessarily categorise the paper as “n/a". Instead, we read the entire paper, and if a study deals with a particular type of testing, e.g., testing of the grammar of a programming language or spreadsheet testing, we placed the paper in the category “others". If the paper lacks any description of the test level, we classified the test level as “n/a".

(1) roles of mutation testing in quality assurance processes:

The first facet concerns the role of mutation testing in quality assurance processes drawn from RQ2.1.1. We identified two classes for the function of mutation testing: assessment and guide. When mutation testing is used as a measure of test effectiveness concerning fault-finding capability, we classify this role as “assessment". While for the “guide" role, mutation testing is adopted to improve the testing effectiveness as guidance, i.e., it is an inherent part of an approach.

To identify and classify the role of mutation testing, we mainly read the description of mutation testing in the *experiment part* of each paper. If we find the phrases which have the same meanings as “evaluate fault-finding ability" or “assess the testing effectiveness" in a paper, we then classify the paper into the class of “assessment". In particular, when used as a measure of testing effectiveness, mutation testing is usually conducted at the end of the experiment; this means mutation testing is not involved in the generation or execution of test suites. Unlike the “assessment" role, if mutation testing is adopted to help to generate test suites or run test cases, we then classify the paper into the “guide" set. In this case, mutation testing is not used in the final step of the experiment.

(2) quality assurance processes:

The second facet focuses on quality assurance processes. Three attributes are relevant to quality assurance processes: the categories of quality assurance processes (RQ2.1.2), test levels (RQ2.1.3) and testing strategies (RQ2.1.4). To identify the categories of quality assurance processes, we group similar quality assurance processes based on information in title and abstract. The quality assurance processes we identified so far consist of 12 classes: test data generation, test-suite reduction/selection, test strategy evaluation, test case minimisation, test case prioritisation, test oracle, fault localisation, program repairing, development scheme evaluation, model clone detection, model review, and fault tolerance. We classify the papers by reading the description appeared in *title and abstract*.

For test level, the values are based on the concept of test level and the authors' specification. More precisely, we consider five test levels: unit, integration, system, others, and n/a. To characterise the test level, we search the exact words "unit", "integration", "system" in the article, as these four test levels are regular terms and cannot be replaced by other synonyms. If there is no relevant result after searching in a paper, we then classify the paper's test level into "n/a", i.e., no specification regarding the test level. Also, for the paper which is difficult for us to categorise into any of the four phases (e.g., testing of the grammar of a programming language and spreadsheet testing) we mark this situation as "others".

For testing strategies, a coarse-grained classification is adequate to gain an overview of the distribution of testing strategies. We identified five classes according to the test design techniques: structural testing, specification-based testing, similarity-based testing, hybrid testing and others [163, 346]. For the structural testing and specification-based testing classes, we further divided the classes into *traditional* and *enhanced*, based on whether other methods improve the regular testing.

To be classified into the "structural testing" class, the paper should either contain the keywords "structure-based", "code coverage-based" or "white box", or use structural test design techniques, such as statement testing, branch testing, and condition testing. For the "specification-based testing" class, the articles should either contain the keywords "black box", "requirement-based" or "specification-based", or use specification-based test design techniques, such as equivalence partitioning, boundary value analysis, decision tables and state transition testing. The similarity-based method aims to maximise the diversity of test cases to improve the test effectiveness; this technique is mainly based on test case relationship rather than software artefacts. Therefore, similarity-based testing does not belong to either structural testing or specification-based testing.

The grey-box testing combines structural testing and specification testing. Besides, several cases are using static analysis, code review or other techniques which cannot fit the above classes; in such situations, we mark the value as "others".

Furthermore, to classify the "enhanced" version of structural and specification-based testing we rely on whether other testing methods were adopted to improve the traditional testing. For instance, Whalen et al. [362] combined the MC/DC coverage metric with a notion of *observability* to ensure the fault propagation conditions. Papadakis and Malevris [292] proposed an automatic mutation test case generation via dynamic symbolic execution. To distinguish such instances from the traditional structural and specification-based testing, we marked them as "enhanced".

(3) mutation tools used in experiments:

For the mutation tools (derived from RQ2.2.1), we are interested in their types, but also in their availability. Our emphasis on tool availability is instigated to address possible replication of the studies. The values of “Yes" or “No" for the tool availability depends on whether the mutation tool is open to the public. The tool type intends to provide further analysis of the mutation tool, which is based on whether the tool is self-developed and whether the tool itself is a complete mutation testing system. We identified five types of mutation tools: existing, partially-based, self-written, manual and n/a. The “existing” tool must be a complete mutation testing system, while “partially-based" means these tools are used as a base or a framework for mutation testing. The example for “partially-based" tools are EvoSuite [145], jFuzz [193], TrpAutoRepair [299], and GenProg [224]. The self-written tool category represents those tools that have been developed by the authors of the study. The “manual" value means the mutants were generated manually according to mutation operators in the studies. Besides, we defined “n/a" value in addition to the “tool types" attribute; the value of “n/a" marks the situation where lacks of a description of mutation tools including tool names/citations and whether manually generated or not.

(4) mutation operators used in experiments: As for the mutation operators (related to RQ2.2.2), we focus on two attributes: description level and generalised classification. The former is again designed to assess the repeatability issue related to mutation testing. The description degree depends on the way the authors presented the mutation operators used in their studies, consisting of three values: “well-defined", “not sufficient" and “n/a". If the paper showed the complete list of mutation operators, then we classify such a paper into “well-defined". The available full list includes two main situations: (1) the authors listed each name of mutation operator(s) and/or specified how the mutation operators make changes to programs in the articles; (2) the studies adopted existing tools and mentioned the used mutation operator (including the option where all or the default set of mutation operators provided by that tool were used). Thus, the well-defined category enables the traceability of the complete list of mutation operators. Instead, if there is some information about the mutation operators in the article but not enough for the replication of the whole list of mutation operators, then we classify the paper into “not sufficient". The typical example is that the author used such words as “etc.", “such as" or, “e.g." in the specification of the mutation operators; this indicates that only some mutation operators are explicitly listed in the paper, but not all. Finally, we use the label “n/a" when no description of the mutation operators was given in the paper at all.

To compare the mutation operators from *different tools* for *different programming languages*, and to analyse the popularity of involved mutation operators amongst the papers, we collected the information about mutation operators mentioned in the articles. Notably, we only consider the articles which are classified as “well-defined". We excluded the papers with “not sufficient" label as their lists of mutation operators are not complete as this might result in biased conclusions based on incomplete information. Moreover, during the reading process, we found that different mutation testing tools use different naming conventions for their mutation operators. For example, in MuJava [240], the mutation operator which replaces relational operators with other relational operators is called “Relational Operator Replacement", while that is named “Conditionals Boundary Mutator" in PIT [3]. Therefore, we saw a need to compose a generalised classification of mutation operators, which enables us to more easily compare mutation operators from different tools or definitions.

The method we adopted here to generate the generalised classification is to group the similar mutation operators among all the existing mutation operators in the literature based on how they mutate the programs. Firstly, mutation testing can be applied to both program source code and program specification. Thus, we classified the mutation operators into two top-level groups: program mutation and specification mutation operators, in a similar vein to Jia and Harman's survey. As we are more interested in program mutation, we further analysed this area and summarised different mutation operators based on literature. More specifically, we first followed the categories and naming conventions of MuJava [237, 238] and Proteum [43] as their mutation operator groups are more complete than the others. Based on the mutation operator groups from MuJava and Proteum, we further divided the program mutation operators into three subcategories: expression-level, statement-level, and others. The expression-level mutation operators focus on the inner components of the statements, i.e., operators (method-level mutation operators in MuJava [238]) and operands (Constant and Variable Mutations in Proteum [43]), while the statement-level ones mutate at least a single statement (Statement Mutations in Proteum [43]). As we are interested in a more generalised classification independent of the programming language, we came up with the class of "others" to include mutation operators related to the programming language's unique features, e.g., Objected-Oriented specific and Java-specific mutation operators (Class Mutation in MuJava [237]). It is important to note that our generalised classification of mutation operators aims to provide an overall distribution of different mutation operator groups. Thus, we did not look into lower-level categories. If the paper used one of the mutation operators in one group, we assign the group name to the paper. For example, while the PIT tool only adopts a small number of arithmetic operators [3], we still assign PIT with the "arithmetic operator".

Our generalised classification of mutation operators is as follows:

Listing 1 *Generalised classification of mutation operators*

1. *Specification mutation*
2. *Program mutation*
   (a) *Expression-level*
      i. ***arithmetic operator:*** *it mutates the arithmetic operators (including addition "+", subtraction "−", multiplication "*", division "/", modulus "%", unary operators "+", "−", and short-cut operators "++", "−−")*[1] *by replacement, insertion or deletion.*
      ii. ***relational operator:*** *it mutates the relational operators (including ">", ">=", "<", "<=", "==", "!=") by replacement.*
      iii. ***conditional operator:*** *it mutates the conditional operators (including and "&", or "|", exclusive or "^", short-circuit operator "&&", "||", and negation "!") by replacement, insertion or deletion.*

[1]The syntax of these operators might vary slightly in different languages. Here we just used the operators in Java as an example. So as the same in (ii) - (vi) operators.

iv. ***shift operator:*** *it mutates the shift operators (including “>>", “<<" and “>>>") by replacement.*

v. ***bitwise operator:*** *it mutates the bitwise operators (including bitwise and “&", bitwise or “|", bitwise exclusive or “^" and bitwise negation “~") by replacement, insertion or deletion.*

vi. ***assignment operator:*** *it mutates the assignment operators (including the plain operator “=" and short-cut operators “+=", “− =", “*=", “/=", “%=", “&=", “|=", “^=", “<<=", “>>=", “>>>=") by replacement. Besides, the plain operator “=" is also changed to “==" in some cases.*

vii. ***absolute value:*** *it mutates the arithmetic expression by preceding unary operators including ABS (computing the absolute value), NEGABS (compute the negative of the absolute value) and ZPUSH (testing whether the expression is zero. If the expression is zero, then the mutant is killed; otherwise execution continues and the value of the expression is unchanged)*[2].

viii. ***constant:*** *it changes the literal value including increasing/decreasing the numeric values, replacing the numeric values by zero or swapping the boolean literal (`true`/`false`).*

ix. ***variable:*** *it substitutes a variable with another already declared variable of the same type and/or of a compatible type.*[3]

x. ***type:*** *it replaces a type with another compatible type including type casting.*[4]

xi. ***conditional expression:*** *it replaces the conditional expression by `true`/`false` so that the statements following the conditional always execute or skip.*

xii. ***parenthesis:*** *it changes the precedence of the operation by deleting, adding or removing the parentheses.*

(b) *Statement-level*

i. ***`return` statement:*** *it mutates `return` statements in the method calls including `return` value replacement or `return` statement swapping.*

ii. ***`switch` statement:*** *it mutates `switch` statements by making different combinations of the `switch` labels (`case`/`default`) or the corresponding block statement.*

iii. ***`if` statement:*** *it mutates `if` statements including removing additional semicolons after conditional expressions, adding an `else` branch or replacing last `else if` symbol to `else`.*

iv. ***statement deletion:*** *it deletes statements including removing the method calls or removing each statement*[5].

[2]The definition of this operator is from the Mothra[210] system. In some cases, this operator only applies the absolute value replacement.

[3]The types of the variables varies in different programming languages.

[4]The changes between the objects of the parent and the child are excluded which belongs to “OO-specific"

[5]To maintain the syntactical validity of the mutants, semicolons or other symbols, such as `continue` in Fortran, are retained.

v. ***statement swap:*** *it swaps the sequences of statements including rotating the order of the expressions under the use of the* comma *operator, swapping the contained statements in* `if-then-else` *statements and swapping two statements in the same scope.*

vi. ***brace:*** *it moves the closing brace up or down by one statement.*

vii. `goto` ***label:*** *it changes the destination of the* `goto` *label.*

viii. ***loop trap:*** *it introduces a guard (trap after n^{th} loop iteration) in front of the loop body. The mutant is killed if the guard is evaluated the n^{th} time through the loop.*

ix. ***bomb statement:*** *it replaces each statement by a special* `Bomb()` *function. The mutant is killed if the* `Bomb()` *function is executed which ensures each statement is reached.*

x. ***control-flow disruption (break/continue):*** *it disrupts the normal control flow by adding, removing, moving or replacing* `continue`*/*`break` *labels.*

xi. ***exception handler:*** *it mutates the exception handlers including changing the* `throws`*,* `catch` *or* `finally` *clauses.*

xii. ***method call:*** *it changes the number or position of the parameters/arguments in a method call, or replace a method name with other method names that have the same or compatible parameters and result type.*

xiii. `do` ***statement:*** *it replaces* `do` *statements with* `while` *statements.*

xiv. `while` ***statement:*** *it replaces* `while` *statements with* `do` *statements.*

(c) Others

i. ***OO-specific:*** *the mutation operators related to O(bject)-O(riented) Programming features [237], such as Encapsulation, Inheritance, and Polymorphism, e.g.* `super` *keyword insertion.*

ii. ***SQL-specific:*** *the mutation operators related to SQL-specific features [342], e.g. replacing* ***SELECT*** *to* ***SELECT DISTINCT****.*

iii. ***Java-specific***[6]***:*** *the mutation operators related to Java-specific features [237] (the operators in Java-Specific Features), e.g.* `this` *keyword insertion.*

iv. ***JavaScript-specific:*** *the mutation operators related to JavaScript-specific features [252] (including DOM, JQUERY, and XMLHTTPREQUEST operators), e.g.* `var` *keyword deletion.*

v. ***SpreadSheet-specific:*** *the mutation operators related to SpreadSheet-specific features [61], e.g. changing the range of cell areas.*

vi. ***AOP-specific:*** *the mutation operators related to A(spect)-O(riented)-P(rogramming) features [110, 372], e.g. removing pointcut.*

vii. ***concurrent mutation:*** *the mutation operators related to concurrent programming features [182, 183], e.g. replacing* `notifyAll()` *with* `notify()`*.*

viii. ***Interface mutation:*** *the mutation operators related to Interface-specific features [184, 375], suitable for use during integration testing.*

[6]This set of mutation operators originated from Java features but not limited to Java language, since other languages can share certain features, e.g., `this` keyword is also available in C++ and C#, and `static` modifier is supported by C and C++ as well.

(5) description of the equivalent mutant problem & (6) description of cost reduction techniques for mutation testing:

The fifth and sixth facets aim to show how the most significant problems are coped with when applying mutation testing (related to RQ2.2.3 and RQ2.2.4 respectively). We composed the list of techniques based on both our prior knowledge and the descriptions given in the papers. We identified seven methods for dealing with the equivalent mutant problem and five for reducing computational cost except for “n/a" set (more details are given in Table 2.3).

For the equivalent mutant problem, we started by searching the keywords “equivalen*" and “equal" in each paper to target the context of the equivalent mutant issue. Then we extracted the corresponding text from the articles. If there are no relevant findings in a paper, we mark this article as “n/a" which means the authors did not mention how they overcame the equivalent mutant problem. Here, it should be noted that we only considered the description related to the equivalent mutant problem given by the authors; this means we excluded the internal heuristic mechanisms adopted by the existing tools if the author did not point out such internal approaches. For example, the tool of JAVALANCHE [313] ranks mutations by impact to help users detect the equivalent mutants. However, if the authors who used JAVALANCHE did not specify that internal feature, we do not label the paper into the class that used the approach of “ranking the mutations".

For the cost reduction techniques, we read the experiment part carefully to extract the reduction mechanism from the papers. Also, we excluded runtime optimisation and selective mutation. The former one, runtime optimisation, is an internal optimisation adopted during the tool implementation, therefore such information is more likely to be reported in the tool documentation. We did not consider the runtime optimisation to avoid incomplete statistics. As for the second one, selective mutation, we assume it is adopted by all papers since it is nearly impossible to implement and use all the operators in practice. If a paper does not contain any description of the reduction methods in the experiment part, we mark this article as “n/a".

(7) subjects involved in the experiment:

For the subject programs in the evaluation part, we are interested in three aspects: programming language, size, and data availability. From the programming language, we can obtain an overall idea of how established mutation testing is in each programming language domain and what the current gap is. From the subject size, we can see the scalability issue related to mutation testing. From the data availability situation, we can assess the replicability of the studies.

For the programming language, we extracted the programming language of the subjects involved in the experiment in these articles, such as Java, C, SQL, etc. If the programming language of the subject programs is not clearly pointed out, we mark it as “n/a". Note, more than one languages might be involved in a single experiment.

For the subject size, we defined four categories according to the lines of code (LOC): preliminary, small, medium and large. If the subject size is less than 100 LOC, then we classify it into the “preliminary" category. If the size is between 100 to 10K LOC, we consider it “small", while between 10K and 1M LOC we appraised it as “medium". If the size is greater than 1M LOC, we consider it as “large". Since our size scale is based on LOC, if the LOC of the subject is not given, or other metrics are used, we mark it as “n/a”. To assign the value to paper, we always take the *biggest* subjects used in the papers.

For the data available, we defined two classes: Yes and No. “Yes" means *all* subjects in the experiments can be openly accessible; this can be identified either from the keywords “open source", `SIR` [126], `GitHub`[7], `SF100` [146] or `SourceForge`[8], or from the open link provided by the authors. It is worth noting that if one of the subjects used in a study is not available, we classify the paper as “No".

The above facets of interest and corresponding attributes and detailed specification of values are listed in Table 2.3.

Facet	**Attribute**	**Value**	**Description**
Roles	classification	assessment	assessing the fault-finding effectiveness
		guide	improving other quality assurance processes as guidance
Quality assurance processes	category	test data generation	creating test input data
		test-suite reduction/selection	reducing the test suite size while maintaining its fault detection ability
		test strategy evaluation	evaluating test strategies by carrying out the corresponding whole testing procedure, including test pool creation, test case selection and/or augmentation and testing results analysis.
		test-case minimisation	simplifying the test case by shortening the sequence and removing irrelevant statements
		test case prioritisation	reordering the execution sequence of test cases
		test oracle	generating or selecting test oracle data
		fault localisation	identifying the detective part of a program given the test execution information
		program repairing	generating patches to correct detective part of a program
		development scheme evaluation	evaluating the practice of software development process via observational studies or controlled experiments, such as Test-Driven Development (TDD)
		model clone detection	identifying similar model fragments within a given context
		model review	determining the quality of the model at specification level using static analysis techniques
		fault tolerance	assessing the ability of the system to continue operating properly in the event of failure
	test level	unit	quality assurance processes focus on unit level. A typical example of unit testing includes: using unit testing tools, such as Junit and Nunit, intra-method testing, intra-class testing.
		integration	quality assurance processes focus on integration level. A typical example of integration testing includes: caller/callee and inter-class testing
		system	quality assurance processes focus on system level. A typical examples of system testing include: high-level model-based testing techniques and high-level specification abstraction methods
		others	quality assurance processes are not related to source code. A typical example includes: grammar.
		n/a	no specification about the testing level in the article.
	testing strategy	structural	white-box testing uses the internal structure of the software to derive test cases, such as statement testing, branch testing, and condition testing
		enhanced structural	adopting other methods to improve the traditional structural testing, mutation-based techniques, information retrieval knowledge, observation notations and assertion coverage
		specification-based	viewing software as a black box with input and output, such as equivalence partitioning, boundary value analysis, decision tables and state transition testing
		enhanced specification-based	adopting other methods to improve the traditional specification-based testing, such as mutation testing.
		similarity-based	maximising the diversity of test cases to improve the test effectiveness
		grey-box	combining structural testing and specification testing together
		others	using static analysis, or focusing on other testing techniques which cannot fit in above six classes
Mutation Tools	availability	Yes/No	Yes: open to the public; No: no valid open access
	type	existing tool	a complete mutation testing system
		partially-based	used as a base or framework for mutation testing
		self-written	developed by the authors and the open link of the tool is also accessible
		manual	generating mutants manually based on the mutation operators
		n/a	no description of the adopted mutation testing tool
Mutation Operators	description Level	well-defined	the complete list of mutation operators is available
		not sufficient	the article provides some information about mutation operators but the information is not enough for replication

[7]https://github.com/
[8]https://sourceforge.net/

		n/a	no description of the mutation operators
	generalised classification	refer to Listing 1	refer to Listing 1
Equivalence Solver	methods	not killed as equivalent not killed as non-equivalent no investigation manual model checker reduce likelihood deterministic model n/a	treating mutants not killed as equivalent treating mutants not killed as non-equivalent no distinguishing between equivalent mutants and non-equivalent ones manual investigation using model checker to remove functionally equivalent mutants generating mutants that are less likely to be equivalent, such as using behaviour-affecting variables, carefully-designed mutation operators, and constraints binding adopting the deterministic model to make the equivalence problem decidable no description of mutant equivalence detector
Reduction Technique	methods	mutant sample fixed number weak mutation higher-order selection strategy n/a	randomly select a subset of mutants for testing execution based on fixed selection ratio select a subset of mutants based on a fixed number compare internal state of the mutant and the original program immediately after the mutated statement(s) reduce the number of mutants by selecting higher-order mutants which contain more than one faults generate fewer mutants by selecting where to mutate based on a random algorithm or other techniques no description of reduction techniques (except for runtime optimisation and selective mutation)
Subject	language	Java, C, C#, etc.	various programming languages
	size (maximum)	preliminary small medium large n/a	<100 LOC 100 ~ 10K LOC 10K ~ 1M LOC > 1M LOC no description of program size regarding LOC
	availability	Yes/No	Yes: open to the public; No: no valid open access

Table 2.3: Attribute framework

2.3.4. Review Protocol Validation

The review protocol is a critical element of a systematic literature review and researchers need to specify and carry out procedures for its validation [85]. The validation procedure aims to eliminate the potential ambiguity and unclear points in the review protocol specification. In this review, we conduct the review protocol validation among the three authors. We also used the results to improve our review protocol. The validation focuses on selection criteria and attribute framework, including the execution of two pilot runs of study selection procedure and data extraction process.

Selection Criteria Validation.

We performed a pilot run of the study selection process, for which we randomly generated ten candidate papers from selected venues (including articles out of our selection scope) and carried out the paper selection among the three authors independently based on the inclusion/exclusion criteria. After that, the three authors compared and discussed the selection results. The results show that for 9 out of 10 papers, the authors had an immediate agreement. The three authors discussed the one paper that showed disagreement, leading to a revision of the first inclusion/exclusion criterion. In the first exclusion criterion, we added “solely” to the end of the sentence “...This criterion excludes the research on mutation testing itself...". By adding “solely" to the first criterion, we include articles whose main focus is mutation testing, but also cover the application of mutation testing.

Attribute Framework Validation.

To execute the pilot run of the data extraction process, we randomly select ten candidate papers from our selected collection. These 10 papers are classified by all three authors independently using the attribute framework that we defined earlier. The discussion that follows from this process leads to revisions of our attribute framework. Firstly, we clarified that the information extracted from the papers must have the same meaning as described by the authors; this mainly means that we cannot further interpret the information. If the article does not provide any clear clue for a certain attribute, we use the phrase "not specified" ("n/a") to mark this situation. By doing so, we minimise the potential misinterpretation of the articles.

Secondly, we ensured that the values of the attribute framework are as complete as possible so that for each attribute we can always select a value. For instance, when extracting quality assurance processes information from the papers, we can simply choose one or several options of the 12 categories provided by the predefined attribute framework. The purpose of providing all possible values to each attribute is to assist data extraction in an unambiguous and trustworthy manner. Through looking at the definitions of all potential values for each attribute, we can easily target unclear or ambiguous points in data extraction strategy. If there are missing values for certain attributes, we can only add the additional data definition to extend the framework. The attribute framework can also be of clear guideline for future replication. Furthermore, we can then present quantitative distributions for each attribute in our later discussion to support our answers to research questions.

To achieve the above two goals, we made revisions to several attributes as well as values. The specified modifications are listed as follows:

Mutation Tools: Previously, we combined tool availability and tool types by defining three values: self-written, existing and not available; this is not clear to distinguish available tools from unavailable ones. Therefore, we further defined two attributes, i.e., tool availability and tool types.

Mutation Operators: We added "description level" to address the interest of how mutation operators are specified in the articles; this also helps in the generalisation of mutation operator classification.

Reduction Techniques: We added the "fixed number" value to this attribute which means the fixed number of selected mutants.

Subjects: We changed the values of "data availability" from "open source", "industrial" or "self-defined" to "Yes" or "No". Since the previous definitions cannot distinguish between available and unavailable datasets.

2.4. Review Results

After developing the review protocol, we conducted the task of article characterisation accordingly. Given the attribute assignment under each facet, we are now at the stage of interpreting the observations and reporting our results. In the following section, we discuss our review results following the sequence of our research questions. While Section 2.4.1 deals with the observations related to how mutation testing is applied (RQ2.1), Section 2.4.2 will present the RQ2.2-related discussion. For each sub-research question, we will first show the distribution of the relevant attributes and our interpretation of the

Table 2.4: Summary of quality assurance processes

Testing Activity	**No. of papers classified as “assessment"**	**No. of papers classified as “guide"**	**No. of papers**
test data generation	38	36	75
test strategy evaluation	63	6	70
test oracle	13	5	18
test case prioritisation	11	6	17
test-suite selection/reduction	10	5	15
fault localisation	8	4	12
program repairing	2	1	3
test case minimisation	1	1	2
fault tolerance	1	0	1
development scheme evaluation	0	1	1
model clone detection	1	0	1
model review	1	0	1
Total	134	57	191

results (marked as **Observation**). Each answer to a sub-research question is also summarised at the end. More detailed characterisations results of all the surveyed papers are presented in our GitHub repository [23].

2.4.1. RQ2.1: How is MT used in quality assurance processes?

RQ2.1.1 & RQ2.1.2: Which role does mutation testing play in each quality assurance process?

Observation.

We opt to discuss the two research questions RQ2.1.1 and RQ2.1.2 together, because it gives us the opportunity to analyse per quality assurance (e.g., test data generation) whether mutation testing is used as a way to guide the technique, or whether mutation testing is used as a technique to assess some (new) approach. Consider Table 2.4, in which we report the role mutation testing plays in the two columns “Assessment" and “Guide" (see Table 2.3 for the explanation about our attribute framework), while the quality assurance processes are projected onto the rows. The table is then populated with our survey results, with the additional note that some papers belong to multiple categories.

As Table 2.4 shows, test data generation and test strategy evaluation occupy the majority of quality assurance processes (accounting for 75.9%) and test suite reduction/selection (7.9%). Only two instances studied test-case minimisation; this shows mutation testing has not been widely used to simplify test cases by shortening the test sequence and removing irrelevant statements.

As the two roles (assessment and guide) are used quite differently depending on the quality assurance processes, we will discuss them separately. Also, for the “guide” role, for which we see an increasing number of applications in recent decades, we find a number of hints and insights for future researchers to consider, which explains why we will analyse this part in a more detailed way when compared to the description of mutation testing as a tool for assessment.

(1) Assessment.

We observed that mutation testing mainly serves as an assessment tool to evaluate the fault-finding ability of the corresponding test or debug techniques (70.2%) as it is widely considered as a "high end" test criterion [47]. To this aim, mutation testing typically generates a significant number of mutants of a program, which are sometimes also combined with natural defects or hand-seeded ones. The results of the assessment are usually quantified as metrics of fault-finding capability: mutation score (or mutation coverage, mutation adequacy) [51, 53] and killed mutants [76, 107] are the most common metrics in mutation testing. Besides, in test-case prioritisation, the Average Percentage of Faults Detected (APFD) [52, 306], which measures the rate of fault detection per percentage of test suite execution, is also popular.

Amongst the papers in our set, we also found 19 studies that performed *mutant analysis*, which means that the researchers tried to get a better understanding about mutation faults, e.g., which faults are more valuable in a particular context. A good example of this mutant analysis is the hard-mutant problem investigated by Czemerinski et al. [108] where they analysed the failure rate for the hard-to-kill mutants (killed by less than 20% of test cases) using the domain partition approach.

(2) Guide.

To provide insight into how mutation testing acts as guidance to improve testing methods per quality assurance process, we will highlight the most significant research efforts to demonstrate why mutation testing can be of benefit as a building block to guide other quality assurance processes. In doing so, we hope the researchers in this field can learn from the current achievements so as to explore other interesting applications of mutation testing in the future.

Firstly, let us start with test data generation, which attracts most interest when mutation testing is adopted as a building block (36 instances). The main idea of mutation-based test data generation is to generate test data that can effectively kill mutants. For automatic test data generation, killing mutants serves as a condition to be satisfied by test data generation mechanisms, such as constraint-based techniques and search-based algorithms; in this way, mutation-based test data generation can be transformed into the structural testing problem. The mutation killable condition can be classified into three steps as suggested by Offutt and Untch [277]: reachability, necessity, and sufficiency. When observing the experiments contained in the papers that we surveyed (except the model-based testing), we see that with regard to the killable mutant condition most papers (73.3%) are satisfied with a weak mutation condition (necessity), while a strong mutation condition (sufficiency) appears less (33.3%). The same is true when comparing first-order mutants (93.3%) to higher-order mutants (6.7%). Except for the entirely automatic test data generation, Baudry et al.[67–69] focused on the automation of the test case enhancement phase: they optimised the test cases regarding mutation score via genetic and bacteriological algorithms, starting from an initial test suite. Von Mayrhauser et al.[356] and Smith and Williams[324] augmented test input data using the requirement of killing as many mutants as possible. Compared to the existing literature survey of Hanh et al. [170], which shed light on mutation-based test data generation, we cover more studies and extend their work to 2015.

The second-most-frequent use cases when applying mutation testing to guide the testing efforts come from test case prioritisation (6 instances) and test strategy evaluation (6 instances). For test case prioritisation, the goal is to detect faults as early as possible in the regression testing process. The incorporation of measures of fault-proneness into prioritisation techniques is one of the directions to overcome the limitation of the conventional coverage-based prioritisation methods. As relevant substitutes of real faults, mutants are used to approximate the fault-proneness capability to reschedule the testing sequences. Qu et al. [300] ordered test cases according to prior fault detection rate using both hand-seeded and mutation faults. Kwon et al. [221] proposed a linear regression model to reschedule test cases based on Information Retrieval and coverage information, where mutation testing determines the coefficients in the model. Moreover, Rothermel et al. [306, 307] and Elbaum et al. [131] compared different approaches of test-case prioritisation, which included the prioritisation in order of the probability of exposing faults estimated by the killed mutant information. In Qi et al. [299]'s study, they adopted a similar test-case prioritisation method to improve patch validation during program repairing.

Zooming in on the test strategy evaluation (6 instances), we observe, on the one hand, the idea of incorporating an estimation of fault-exposure probability into test data adequacy criteria intrigued some researchers. Among them are Chen et al. [97]: in their influential work they examined the fault-exposing potential (FEP) coverage adequacy which is estimated by mutation testing with four software testers to explore the cost-effectiveness of mutation testing for manually augmenting test cases. Their results indicate that mutation testing was regarded as an effective but relatively expensive technique for writing new test cases.

When it comes to the test oracle problem (5 instances), mutation testing can also be of benefit for driving the generation of assertions, as the prerequisite for killing the mutant is to distinguish the mutant from the original program. In Fraser and Zeller [150]'s study, they illustrated how they used mutation testing to generate test oracles: assertions, as commonly used oracles, are generated based on the trace information of both the unchanged program and the mutants recorded during the executions. First, for each difference between the runs on the original program and its mutants, the corresponding assertion is added. After that, these assertions are minimised to find a sufficient subset to detect all the mutants per test case; this becomes a minimum set covering problem. Besides, Staats et al. [326] and Gay et al. [155] selected the most “effective" oracle data by ranking variables (trace data) based on their killed mutants information.

Mutation-based test-suite reduction (5 instances) relies on the number of killed mutants as a heuristic to perform test-suite reduction, instead of the more frequently used traditional coverage criteria, e.g., statement coverage. The intuition behind this idea is that the reduction based on the mutation faults can produce a better-reduced test suite with less or no loss in fault-detection capability. The notable examples include an empirical study carried out by Shi et al. [319] who compared the trade-offs among various test-suite reduction techniques based on statement coverage and killed mutants.

As for the fault localisation (4 instances), the locations of mutants are used to assist the localisation of “unknown" faults (the faults which have been detected by at least one test case, but that have still to be located [290]). The motivation of this approach is based on the following observation: “Mutants located on the same program statements frequently exhibit a similar behaviour" [290]. Thus, the identification of an “unknown" fault could be obtained thanks to a mutant at the same (or close) location. Taking advantage of the implicit link between the behaviour of “unknown" faults with some mutants, Murtaza et al. [260] used the traces of mutants and prior faults to train a decision tree to identify the faulty functions. Also, Papadakis et al. [290, 291] and Moon et al. [253] ranked the suspiciousness of “faulty" statements based on their passing and failing test executions of the generated mutants.

From the aforementioned guide roles of the quality assurance processes, we can see that mutation testing is mainly used as an indication of the potential defects: either (1) to be killed in test data generation, test case prioritisation, test-suite reduction, and test strategy evaluation, or (2) to be suspected in the fault localisation. In most cases, mutation testing serves as *where*-to-check constraints, i.e., introducing a modification in a certain statement or block. In contrast, only four studies applied mutation testing to solving the test oracle problem, which targets the *what*-to-check issue. The *what*-to-check problem is not an issue unique to mutation testing, but rather an inherent challenge of test data generation. As mentioned above, mutation testing cannot only help in precisely targeting at *where* to check but also suggesting *what* to check for [150] (see the first recommendation labelled as **R1** in Section 2.4.4). In this way, mutation testing could be of benefit to improve the test code quality.

After we had analysed how mutation testing is applied to guide various quality assurance processes, we are now curious to better understand how these mutation-based testing methods were evaluated, especially because mutation testing is commonly used as an assessment tool. Therefore, we summed up the evaluation fault types among the articles labelled as “guide" in Table 2.5. We can see 43 cases (75.4%), which is the addition of the first and the third rows in Table 2.5 (39+ 4), still adopted mutation faults to assess the effectiveness. Among these studies, four instances [155, 172, 232, 326] realised the potentially biased results caused by the same set of mutants being used in both guidance and assessment. They partitioned the mutants into different groups and used one as evaluation set. Besides, one study [382] used a different mutation tool, while the other [184] adopted different mutation operators to generate mutants intending to eliminate bias. These findings signal an open issue: how to find an adequate fault set instead of mutation faults to effectively evaluate mutation-based testing methods? (see the second recommendation labelled as **R2** in Section 2.4.4) Although hand-seeded faults and real bugs could be an option, searching for such an adequate fault set increases the difficulty when applying mutation testing as guidance.

Summary.

Test data generation and test strategy evaluation occupy the majority of quality assurance processes when applying mutation testing (75.9%). While as guidance, mutation testing is primarily used in test data generation (36 instances), test strategy evaluation

Table 2.5: Summary of guide role

Evaluation Fault Type	Number of papers
mutation faults	39
hand-seeded faults	7
hand-seeded + mutation faults	4
no evaluation	4
real faults	2
other coverage criteria	1

Table 2.6: Summary of test level

Test Level	Number of papers
n/a	84
unit	77
integration	15
other	10
system	10

(6 instances) and test-case prioritisation (6 instances). From the above observations, we draw one open issue and one recommendation for the “guide" role of mutation testing. The open issue is how to find an adequate fault set instead of mutation faults to effectively evaluate mutation-based testing methods. The recommendation is that mutation testing can suggest not only *where* to check but also *what* to check. *Where* to check is widely used to generate killable mutant constraints in different quality assurance processes, while *what* to check is seldom adopted to improve the test data quality.

RQ2.1.3: Which test level does mutation testing target?

Observations.

Table 2.6 presents the summary of the test level distribution across the articles. We observe that the authors of 84 papers do not provide a clear clue about the test level they target (the class marked as “n/a”). For example, Aichernig et al.[44]’s study: they proposed a fully automated fault-based test case generation technique and conducted two empirical case studies derived from industrial use cases; however, they did not specify which test level they targeted. One good instance that clearly provides the information of the test level is Jee et al.[194] where they specified that their test case generation technique for FBD programs is at unit testing level. This is an open invitation for future investigations in the area to be clearer about the essential elements of quality assurance processes such as the test level. For the remainder of our analysis of RQ2.1.3, we excluded the papers labelled as “n/a" when calculating percentages, i.e., our working set is 107 (191 − 84) papers.

Looking at Table 2.6, mutation testing mainly targets the unit-testing level (72.0%), an observation which is in accordance with the results of Jia and Harman’s survey [197]. One of the underlying causes for the popularity of the unit level could be the origin of mutation testing. The principle of mutation testing is to introduce small syntactic changes to the original program; this means the mutation operators only focus on small parts of the program, such as arithmetical operators and *return* statements. Thus, such small changes mostly reflect the abnormal behaviour of unit-level functionality.

While unit testing is by far the most observed test level category in our set of papers, higher-level testing, such as integration testing (15 instances), can also benefit from the application of mutation testing. Here we highlight some research works as examples: Hao et al. [171] and Do and Rothermel [128] used the programs with system test cases as their subjects in case studies. Hou et al. [184] studied interface-contract mutation in support of integration testing under the context of component-based software. Li et al. [229] proposed a two-tier testing method (one for integration level, the other for system level) for graphical user interface (GUI) software testing. Rutherford et al. [310] defined and evaluated adequacy criteria under system-level testing for distributed systems. In Denaro et al. [120]'s study, they proposed a test data generation approach using data flow information for inter-procedural testing of object-oriented programs.

The important point we discovered here is that all the aforementioned studies did not restrict mutation operators to model integration errors or system ones. In other words, the traditional program mutations can be applied to higher-level testing. Amongst these articles, the mutation operators adopted are mostly at the unit level, e.g., Arithmetic Mutation Replacement, Relational Mutation Replacement. The mutation operators designed for higher-level testing, e.g., [112, 243], are seldom used in these studies. The only three exceptions in our collections are Flores and Polo[141], Vincenzi et al.[349] and Flores and Polo[142], who adopted interface mutation to evaluate the integration testing techniques. This reveals a potential direction for future research: the cross-comparison of different levels of mutation operators and quality assurance processes at different test levels (see the third recommendation labelled as **R3** in Section 2.4.4). The investigation of different levels of mutants can explore the effectiveness of mutation faults at different test levels, such as the doubts whether the integration-level mutation is better than unit-level mutation when assessing testing techniques at the integration level. In the same vein, an analysis of whether mutants are a good alternative to real/hand-seeded ones (proposed by Andrews et al.[50]) at higher levels of testing also seems like an important avenue to check out.

In addition, we created a class "others" in which we list 9 papers that we found hard to classify in any of the other four test phases. These works can be divided into three groups: grammar-based testing [75, 95, 176], spreadsheet-related testing [61, 179, 180] and SQL-related testing [207, 249, 343]. The application of mutation testing on the "other" set indicates that the definition of mutation testing is actually quite broad, thus potentially leading to even more intriguing possibilities [279]: what else can we mutate?

Summary.

The application of mutation testing is mostly done at the unit-level testing (44.0% of papers did not clearly specify their target test level(s)). For reasons of clarity, understandability and certainly replicability, it is very important to understand exactly at what level the quality assurance processes take place. It is thus a clear call to arms to researchers to better describe these essential testing activity features.

Table 2.7: Summary of testing strategies

Testing Strategies	Number of papers
structural testing	57
specification-based testing	52
structural testing (enhanced)	52
others	21
specification-based testing (enhanced)	13
hybrid testing	7
similarity-based testing	3

RQ2.1.4: Which testing strategies does mutation testing support?

Observations.

In Table 2.7 we summarised the distribution of testing strategies based on our coarse-grained classification (e.g., structural testing, specification-based testing) as mentioned in Table 2.3. Looking at Table 2.7, structure-based testing (including the first and the third rows in Table 2.7), 109 instances). The underlying cause could be that structural testing is still the main focus of testing strategies in the software testing context. The other testing strategies have also been supported by mutation testing: (1) specification-based testing (including the second and the fifth rows in Table 2.7) accounts for 65 cases; (2) hybrid testing (combination of structural and structure-based testing) for seven instances, e.g., Briand et al. [86] investigated how data flow information can be used to improve the cost-effectiveness of state-based coverage criteria; (3) three cases applying mutation testing in similarity-based testing; (4) 21 instances in others, e.g., static analysis.

One interesting finding is that enhanced structural testing ranks third, including mutation-based techniques, information retrieval knowledge, observation notations and assertion coverage. The popularity of enhanced structural testing reveals the awareness of the shortage of conventional coverage-based testing strategies has increased.

Compared to enhanced structural testing, enhanced specification-based testing did not attract much interest. The 13 instances mainly adopted mutation testing (e.g., Qu et al. [300] and Papadakis et al. [287]) to improve the testing strategies.

Summary.

Mutation testing has been widely applied in support of different testing strategies. From the observation, the testing strategies other than white box testing can also benefit from the application of mutation testing, such as specification-based testing, hybrid testing, and similarity-based testing. However, structural testing is more popular than the others (57.1%). Moreover, techniques like mutation-based techniques and information retrieval knowledge are also being adopted to improve the traditional structural-based testing, which typically only relies on the coverage information of software artefacts; this serves an indication of the increasing realisation of the limitations of coverage-based testing strategies.

Table 2.8: Summary of mutation tool

Availability	Types	Number of papers	
Yes	existing	96	103
	partially-based	7	
	self-written	1	
No	n/a (no information)	44	92
	existing (given the name/citation)	22	
	self-written	14	
	manual	12	

2.4.2. RQ2.2: How are empirical studies related to mutation testing designed and reported?

RQ2.2.1: Which mutation testing tools are being used?

Observations.

We are interested in getting insight into the types (as defined in Table 2.3) of mutation testing tools that are being used and into their availability. Therefore, we tabulated the different types of mutation testing tools and their availability in Table 2.8. As shown in Table 2.8, 50.3% of the studies adopted existing tools which are open to the public; this matches our expectation: as mutation testing is not the main focus of the studies, if there exists a well-performing and open-source mutation testing tool, researchers are likely willing to use these tools. However, we also encountered 15 cases of using self-implemented tools and 12 studies that manually applied mutation operators. A likely reason for implementing a new mutation testing tool or for manually applying mutation operators is that existing tools do not satisfy a particular need of the researchers. Besides, most existing mutation testing tools are typically targeting one specific language and a specific set of mutation operators [279] and they are not always easy to extend, e.g., when wanting to add a newly-defined mutation operator. As such, providing more flexible mechanisms for creating new mutation operators in mutation testing tools is an important potential direction for future research [211, 279] (see the fourth recommendation labelled as **R4** in Section 2.4.4).

Unfortunately, there are also still 92 studies (48.2%) that do not provide access to the tools, in particular, 44 papers did not provide any accessible information about the tools (e.g., Hong et al.[183], Belli et al.[76] and Papadakis and Malveris[294]), a situation that we marked as “n/a" in Table 2.8. This unclarity should serve as a notice to researchers: the mutation testing tool is one of the core elements in mutation testing, and lack of information on it seriously hinders replicability of the experiments.

Having discussed the tool availability and types, we are wondering which existing open-source mutation testing tools are most popular. The popularity of the tools cannot only reveal their level of maturity, but also provide a reference for researchers entering the field to help them choose a tool. To this end, we summarised the names of mutation tools for different programming languages in Table 2.9. Table 2.9 shows that we encountered 19 mutation tools in total. Most tools target one programming language (except

Table 2.9: Summary of existing mutation tools

Language	**Tool**	**Number of papers**
Java	MuJava/μ-java/Muclipse	41
	JAVALANCHE	9
	MAJOR	9
	PIT/PiTest	7
	Jumble	2
	Sofya	1
	Jester	1
C	Proteum	12
	Proteum/IM	3
	MiLu	2
	SMT-C	1
Fortran	Mothra	4
SQL	SQLMutation/JDAMA	3
	SchemaAnalyst	1
C#	GenMutants	1
	PexMutator	1
JavaScript	MUTANDIS	3
AspectJ	AjMutator	2
UML specification	MoMuT::UML	1

for Mothra [210], which supports both C and Fortran). We encountered seven mutation tools for Java, with the top 3 most-used being MuJava [239], JAVALANCHE [313] and Major [204]. We found that four mutation tools for C are used, where Proteum [113] is the most-frequently applied. Proteum/IM [114] is a special mutation testing tool that targets interface mutation, which concerns integration errors. The integration errors are related to a connection between two units and the interactions along the connection, such as a wrong subprogram call.

In Jia and Harman [197]'s literature review, they summarised 36 mutation tools developed between 1977 and 2009. When comparing their findings (36 tools) to ours (19 tools), we find that there are 12 tools in common. The potential reason for us not covering the other 24 is that we only consider peer-reviewed conference papers and journals; this will likely filter some papers which applied the other 24 mutation tools. Also important to stress, is that the goal of Jia and Harman's survey is different to ours: while we focus on the application of mutation tools, their study surveys articles that introduce mutation testing tools. In doing so, we still managed to discover 8 mutation tools which are not covered by Jia and Harman: (1) two tools are for Java: PIT and Sofya; (2) one for C: SMT-C; (3) one for SQL: SchemaAnalyst; (4) one for UML: MoMuT::UML; (5) two for C#: GenMutants and PexMutator; (6) one for JavaScript: MUTANDIS. Most of these tools

Table 2.10: Summary of description level of mutation operators

Description Level	Number of papers
well-defined	119
n/a	44
not sufficient	28

were released after 2009, which makes them too new to be included in the review of Jia and Harman. Moreover, we can also witness the trend of the development of the mutation testing for programming languages other than Java and C when compared to Jia and Harman [197]'s data.

SUMMARY.
50.3% of the articles that we have surveyed adopt existing (open access) tools, while in a few cases (27 in total) the authors implemented their own tools or seeded the mutants by hand. This calls for a more flexible mutation generation engine that allows to easily add mutation operators or certain types of constraints. Furthermore, we found 44 papers that did not provide any information about the mutation tools they used in their experiments; this should be a clear call to arms to the research community to be more precise when reporting on mutation testing experiments. We have also gained insight into the most popular tools for various programming languages, e.g., MuJava for Java and Proteum for C. We hope this list of tools can be a useful reference for new researchers who want to apply mutation testing.

RQ2.2.2: WHICH MUTATION OPERATORS ARE BEING USED?

OBSERVATIONS
For the mutation operators, we first present the distribution of the three description levels (as mentioned in Table 2.3) in Table 2.10. As Table 2.10 shows, 62.3% (119 instances) of the studies that we surveyed specify the mutation operators that they use, while more than one-third of the articles do not provide enough information about the mutation operators to replicate the studies. These 61 instances are labelled as “n/a" (e.g., Briand et al.[87], DeMillo and Offutt[119] and Shi et al.[320]).

After that, based on our generalised classification of the mutation operators (as defined in Listing 1), we characterised the 119 papers labelled as “well-defined". In addition to the overall distribution of the mutation operators regardless of the programming language, we are also interested in the differences of the mutation operators for different languages as the differences could indicate potential gaps in the existing mutation operator sets for certain programming languages. In Table 2.11 we project the various languages onto seven columns and our predefined mutation operator categories onto the rows, thus presenting the distribution of the mutation operators used in the literature under our research scope.

Overall, we can see that program mutation is more popular than specification mutation from Table 2.11. Among the program mutation operators, the arithmetic, rela-

[1]The Java-specific operator here refers to the `static` modifier change (including insertion and deletion).

Table 2.11: Mutation operators used in our collection

Level	Operator	Java	C	C++	C#	Fortran	SQL	JavaScript	Total
Specification Mutation		2	2	1	-	-	-	-	23
Program Mutation		55	12	4	3	2	5	1	95
Expression-level	arithmetic operator	51	10	4	1	2	3	-	79
	relational operator	47	8	4	1	2	3	-	74
	conditional operator	47	7	4	2	2	3	-	72
	bitwise operator	36	4	2	-	-	-	-	43
	assignment operator	33	4	2	-	-	-	-	39
	constant	18	5	2	-	2	3	-	37
	shift operator	33	2	2	-	-	-	-	36
	variable	15	4	2	-	2	3	1	31
	absolute value	19	3	2	1	1	3	-	31
	conditional expression	9	3	-	1	-	-	-	14
	parenthesis	1	2	-	-	-	-	-	3
	type	-	3	-	-	-	-	-	3
Statement-level	statement deletion	11	4	-	2	2	-	-	23
	method call	11	-	2	-	-	-	1	16
	`return` statement	10	3	-	-	2	-	-	16
	control-flow disruption	6	2	2	-	-	-	-	9
	exception handler	1	-	1	-	-	-	-	5
	`goto` label	-	3	-	-	2	-	-	6
	statement swap	2	2	2	-	-	-	-	5
	bomb statement	-	2	-	-	2	-	-	5
	`switch` statement	2	3	-	-	-	-	-	5
	`do` statement	-	2	-	-	2	-	-	5
	brace	-	3	-	-	-	-	-	3
	loop trap	-	3	-	-	-	-	-	3
	`while` statement	-	3	-	-	-	-	-	3
	`if` statement	-	-	-	-	-	-	-	-
Others	OO-specific	23	-	-	-	-	-	-	26
	Java-specific	17	-	1[1]	-	-	-	-	17
	Interface mutation	4	2	-	-	-	-	-	7
	SQL-specific	-	-	-	-	-	5	-	5
	Concurrent mutation	4	-	-	-	-	-	-	4
	AOP-specific	-	-	-	-	-	-	-	3
	Spreadsheet-specific	-	-	-	-	-	-	-	2
	JavaScript-specific	-	-	-	-	-	-	1	1

tional and conditional operators are the top 3 mutation operators. These three operators are often used together in most cases as their total number of applications are similar. The underlying cause of the popularity of these three operators could be that the three operators are among Offutt et al.[271]'s 5 sufficient mutation operators. Moreover, the expression-level operators are more popular than the statement-level ones. As for the statement-level mutation operators, statement deletion, method call, and `return` statement are the top 3 mutation operators.

When we compare the mutation operators used in different languages to our mutation operator categories, we see that there exist differences between different programming languages, just like we assumed. Table 2.11 leads to several interesting findings that reveal potential gaps in various languages (note that Table 2.11 only listed seven programming languages that have been widely used). Moreover, as Jia and Harman [197] also discussed mutation operators in their review, it is interesting to see whether their summary agrees with our work. Therefore, we also compared our findings to Jia and Harman's as follows:

1. For Java, seven mutation operators at the expression and statement level (except `go to` label which is not supported in Java) are not covered by our survey: type, bomb statement, `do` statement, brace, loop trap, `while` statement and `if` statement. Compared to Jia and Harman's survey, Alexander et al. [45, 79]'s design of Java Object mutation cannot be found in our collection.

2. For C, only two operators are not covered by our dataset. The C programming language does not provide direct support for exception handling. There is no article applying mutation operators that target specific C program defects or vulnerabilities surveyed by Jia and Harman, such as buffer overflows [348] and format string bugs [318].

3. For C++, 3 expression-level, 10 statement-level and the OO-specific operators are not used in our dataset. Jia and Harman have not covered mutation operators for C++.

4. For C#, only a limited set of mutation operators are applied based on our dataset. Our collection has no application of OO-specific operators [123] summarised in Jia and Harman's survey.

5. For Fortran, the earliest programming language mutation testing was applied to, the studies in our collection cover a basic set. These mutation operators agree with Jia and Harman's work.

6. For SQL, since the syntax of SQL is quite different from imperative programming languages, only six operators at the expression level and SQL-specific ones are used in our dataset. Compared to Jia and Harman's literature review, a set of mutation operators addressing SQL injection vulnerabilities [317] are not found in our collection.

7. For JavaScript, only three JavaScript-specific mutation operators are adopted in studies we selected. Mutation operators for JavaScript [251] have not been covered by Jia and Harman as the paper introducing them (i.e., Milani et al.[251]) is more recent.

8. For interface mutation, we have found studies solely targeting Java and C in our selection.

The comparison with Jia and Harman's literature review has shown that for most programming languages (except for C++ and JavaScript), a few mutation operators summarised by Jia and Harman have no *actual* applications in our collection. As we only considered 22 venues, we might miss the studies that adopted these mutation operators in other venues. Without regard to the potential threat of missing papers in our dataset, these mutation operators that have no applications in our collection pose interesting questions for further exploration, e.g., why are these operators seldom used by other researchers? What is the difference between these operators and other widely used operators?

Also, from the above findings, we can see that for different languages the existing studies did not cover all the mutation operators that we listed in Table 2.11: some are caused by the differences in the syntax, while the others could point to potential gaps. However, these potential gaps are just the initial results in which we neither did further analysis to chart the syntax differences of these languages nor investigate the possibility of the equivalent mutants caused by our classification. Moreover, for some languages, e.g., JavaScript, the relevant studies are too few to draw any definitive conclusions. We can only say that Table 2.11 can be a valuable reference for further investigations into mutation operator differences and gaps in different programming languages.

Furthermore, our generalised classification of the existing mutation operators can also be of benefit to compare mutation tools in the future. Thereby, we compared the existing mutation testing tools (as listed in Table 2.9) to our mutation operator categories in Table 2.12. Table 2.12 is based on the documentation or manuals of these tools. Here we used the definitions of mutation operator groups from Listing 1 (mainly based on MuJava [237, 238] and Proteum [43]) as the baseline: if there is a possible mutation missing in a group for a mutation testing tool, we marked "*" in Table 2.12. As there exist different syntaxes in different programming languages, we also consider syntactic differences when categorising the mutation operators of different tools. For example, there is no modulus operator "%" in Fortran (but a `MOD` function instead), therefore, when considering the arithmetic mutation operators for Fortran (Mothra), we do not require the modulus operator "%" to be included in Mothra.

It is important to mention that Table 2.9 is not the complete list of all the existing mutation tools that have been published so far; these tools are chosen to investigate how mutation testing supports quality assurance processes. We analyse them here as they are open to public and have been applied by researchers at least once. The analysis of mutation operators in these tools could also be a valuable resource for researchers in mutation testing to consider.

The result shows that none of the existing mutation testing tools we analysed can cover all types of operators we classified. For seven mutation testing tools for Java, they mainly focus on the expression-level mutations and only five kinds of statement-level mutators are covered. Furthermore, MuJava, PIT, and Sofya provide some OO-specific operators, whereas PIT only supports one type, the Constructor Calls Mutator. For the four mutation testing tools for C (including Mothra) that we have considered, Proteum covers the most mutation operators. SMT-C is an exceptional case of the traditional mutation testing which targets semantic mutation testing. Proteum/IM is the only mutation tool listed in Table 2.12 that supports interface mutation. For the tools designed for C#, OO-specific operators are not present.

Moreover, when we further analyse the missing mutations in each mutation operator group (marked as "*" in Table 2.12), we found that most tools miss one or several mutations compared to our generalised classification. Particularly for the arithmetic operator, only MuJava and Proteum apply all possible mutations. The other tools that adopt the arithmetic operator all miss Arithmetic Operator Deletion (as defined in MuJava [238]).

Another interesting finding when we compared Table 2.11 and Table 2.12, is that the

Table 2.12: Comparison of mutation operators in existing mutation tools

	MuJava/μ-java/Muclipse [11, 237, 238]	PIT/PiTest [3, 13]	JAVALANCHE [9, 313]	MAJOR [21, 39]	Jumble [19]	Sofya [38]	Jester [255]	Proteum [14, 43]	MiLu [10]	SMT-C [98]	Proteum/IM [15, 114]	Mothra [210]	SQLMutation/JDAMA [342]	SchemaAnalyst [370]	GenMutants [8]	PexMutator [26]	MUTANDIS [252]	AjMutator [111]	MoMuT::UML [216]
Specification Mutation	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√
arithmetic operator	√	√*	√*	√*	√*	√*	-	√	√*	√*	-	√*	√*	-	√*	√*	√*	-	-
relational operator	√	√*	-	√	-	√*	-	√	√	√*	-	√	√	-	√	√	√	-	-
conditional operator	√	√*	√*	√*	√*	√*	-	√	√	-	-	√	√	-	√	√	√*	-	-
assignment operator	√	√*	-	√*	-	-	-	-	√	√*	-	-	-	-	-	-	-	-	-
bitwise operator	√	√*	-	√	√*	-	-	√	√*	-	-	-	-	-	-	-	-	-	-
shift operator	√	√*	-	√	√*	-	-	√	-	-	-	-	-	-	-	-	-	-	-
constant	√*	√	√	√	√	-	√*	√	√	√*	-	√	√	-	-	-	√	-	-
variable	√*	√	√	√*	√*	-	-	√	-	-	-	√	√	-	-	-	√	-	-
absolute value	-	-	√	√	-	-	-	-	√*	-	-	√	√	-	√	√	-	-	-
conditional expression	-	√	√	√	-	-	√	√	-	-	-	-	-	-	-	-	-	-	-
parenthesis	-	-	-	-	-	-	-	√	-	√*	-	-	-	-	-	-	-	-	-
type	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	√*	-	-
statement deletion	√	√*	√*	√*	-	-	-	√	√	-	-	√	-	-	-	-	√*	-	-
method call	-	-	-	√	-	√*	-	-	-	√*	-	-	-	-	-	-	√*	-	-
return statement	-	√	-	√	√	-	-	√	-	-	-	√	-	-	-	-	√	-	-
if statement	-	-	-	-	-	-	-	-	-	√*	-	-	-	-	-	-	-	-	-
exception handler	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
goto label	-	-	-	-	-	-	-	√	-	-	-	√	-	-	-	-	-	-	-
control-flow disruption	-	-	-	√	-	-	-	√	-	-	-	-	-	-	-	-	√	-	-
statement swap	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	√*	-	-
bomb statement	-	-	-	-	-	-	-	√	-	-	-	√	-	-	-	-	-	-	-
switch statement	-	√	-	-	√	-	-	√	-	√	-	-	-	-	-	-	√	-	-
do statement	-	-	-	-	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-
brace	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-	-
loop trap	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-	-
while statement	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-	-	-	-
OO-specific	√	√*	-	-	-	√*	-	-	-	-	-	-	-	-	-	-	-	-	-
Java-specific	√	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
SQL-specific	-	-	-	-	-	-	-	-	-	-	-	-	√	√	-	-	-	-	-
JavaScript-specific	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√	-	-
AOP-specific	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	√	-
interface mutation	-	-	-	-	-	-	-	-	-	-	√	-	-	-	-	-	-	-	-

Note: the entry marked with * means the tool does not provide the full possible mutations. Our summary of the mutation tools is based on the available manuals and open repositories (if they exist) for the tools. If there are different versions of the tools, we only consider the newest one.

`if` statement mutator is not used in literature, but it is supported by SMT-C. This observation indicates that not all the operators provided by the tools are used in the studies when applying mutation testing. Therefore, we zoom in on this finding and investigate whether it is a common case that only a subset of the mutation operators from the existing mutation tools is adopted in studies based on our collection. The result shows that 21 studies out of 54[2]applied a subset of the mutation operators from the existing tools; this reinforces our message of the need for "well-defined" mutation operators when reporting mutation testing studies.

Summary.
For the mutation operators, we focused on two attributes: their description level and a generalised classification across tools and programming languages. When investigating the description level of the mutation operators that are used in studies, we found that 62.3% (119 instances) explicitly defined the mutation operators used; this leads us to strongly recommend improving the description level for the sake of replicability. Furthermore, the distribution of mutation operators based on our predefined categories shows the lacking of certain mutation operators in some programming languages among the existing (and surveyed) mutation testing tools. A possible avenue for future work is to see which of the *missing* mutation operators can be implemented for the programming languages lacking these operators.

RQ2.2.3: Which approaches are used to overcome the equivalent mutant problem when applying mutation testing?

Observations.
In Table 2.13, we summarised our findings of how the studies that we surveyed deal with the equivalent mutant problem. More specifically, Table 2.13 presents how many times we encountered each of the approaches for solving the equivalent mutant problem. When looking at the results, we first observe that in 56.5% of the cases we assigned “n/a”, such as Androutsopoulos et al.[52] and Flores and Polo[142].

As shown in Table 2.13, there are only 17 instances actually adopting equivalent mutant detectors by using automatic mechanisms. Specifically, 6 instances use “model checker" (e.g., Gay et al. [156]); 8 instances use “reduce likelihood" (e.g., Milani et al. [251]); and 3 instances apply “deterministic model" (Belli et al.[76]). In the remaining papers, the problem of equivalent mutants is solved by: (1) manual analysis (e.g., Liu et al. [231] and Xie et al. [371]); (2) making assumptions (treating mutants not killed as either equivalent or non-equivalent, e.g., Fang et al.[136] and Rothermel et al. [307]); (3) no investigation (e.g., Offutt and Liu [273], Chen et al [97] and Fraser and Zeller [150]). The manual investigation (38 instances) and the method of treating mutants not killed as equivalent (17 instances) are more commonly used than other methods. We also compared our results with Madeyski et al. [241]'s survey on the equivalent mutant problem. In their review, they reviewed 17 methods for overcoming the equivalent mutant problem. Among these 17 techniques, we found that only the model-checker approach [129] was adopted.

We can only speculate as to the reasons behind the situation above: Firstly, most studies use mutation testing as an evaluation mechanism or guiding heuristic, rather than their main research topic. So, the authors might be not willing to spare too much effort in dealing with problems associated with mutation testing. Moreover, looking at the internal features of existing tools used in literature (in Table 2.14), we found that only five tools adopt techniques to address the equivalent mutant problem. Most of the tools did not assist in dealing with the equivalent mutant problem. Therefore, in this chapter, we consider the aforementioned three solutions: (1) manual analysis, (2) making assumptions, or (3) no investigation. If there exists a well-developed auxiliary tool that can

[2]The studies that are categorised as “yes" in the tool availability, “existing tool" in the tool type and “well-defined" in the mutation operator description level.

Table 2.13: Summary of methods for overcoming E(quivalent) M(utant) P(roblem)

Equivalence Detector	**Number of papers**
n/a	108
manual investigation	38
not killed as equivalent	17
no investigation	11
reduce likelihood	8
model checker	6
deterministic model	3
not killed as non-equivalent	3

Table 2.14: Inner features of existing mutation tools

Language	**Tool**	**Equivalent Mutants**	**Cost Reduction**
Java	MuJava/μ-java/Muclipse	n/a	MSG, bytecode translation (BCEL) [239]
	PIT/PiTest	n/a	Bytecode translation (ASM), coverage-based test selection [24]
	JAVALANCHE	Ranking mutations by impact [313]	MSG, bytecode translation (ASM), coverage-based test selection, parallel execution [313]
	MAJOR	n/a	Compiler-integrated, coverage-based test selection [204]
	Jumble	n/a	Bytecode translation (BCEL), conventional test selection [190]
	Sofya	n/a	Bytecode translation (BCEL) [256]
	Jester	n/a	n/a
C	Proteum	n/a	Mutant sample [113]
	MiLu	Trivial Compiler Equivalence [288]	Higher-order mutants, test harness [196]
	SMT-C	n/a	Interpreter-based, weak mutation [109]
	Proteum/IM	n/a	Compiler-based, control flow optimisation [114]
Fortran	Mothra	n/a	Interpreter-based [197]
SQL	SQLMutation/JDAMA	Constraint binding [342]	n/a
	SchemaAnalyst	n/a	n/a
C#	GenMutants	n/a	n/a
	PexMutator	n/a	Compiler-based [382]
JavaScript	MUTANDIS	Reduce likelihood [252]	Selection strategy [252]
AspectJ	AjMutator	Static analysis [111]	Compiler-based [111]
UML specification	MoMuT::UML	n/a	Compiler-based [216]

Note: "n/a" in the table means we did not find any relevant information recorded in literature or websites, and some tools might adopt certain techniques but did not report such information in the sources we can trace.

be seamlessly connected to the existing mutation systems for helping the authors detect equivalent mutants, this tool might be more than welcomed. We recommend that future research on the equivalent mutant problem can further implement their algorithms in such an auxiliary tool and make it open to the public (see the fifth recommendation labelled as **R5** in Section 2.4.4).

Secondly, the mutation score is mainly used as a relative comparison for estimating the effectiveness of different techniques. Sometimes, mutation testing is only used to generate likely faults; equivalent mutants have no impact on the other measures such as the Average Percentage of Fault Detection rate (APFD) [306]. Furthermore, the definition of the mutation score is also modified by some authors (e.g., Rothermel et al.[307]) : they used the total number of mutants as the denominator instead of the number of non-equivalent mutants. The equivalent mutant problem seems to not pose a significant threat to the validation of the testing techniques involved in these studies.

However, we should not underestimate the impact of the equivalent mutant problem on the accuracy of the mutation score. Previous empirical results indicated that 10 to 40 percent of mutants are equivalent [270, 274]. What's more, Schuler and Zeller [314] further claimed that around 45% of all undetected mutants turned out to be equivalent; this observation leads to the assumption that by simply treating mutants not killed as equivalent mutations, we could be overestimating the mutation score. Therefore, we recommend performing more large-scale investigations on whether the equivalent mutant problem has a strong impact on the accuracy of the mutation score.

Summary.

The techniques for equivalent mutant detection are not commonly used when applying mutation testing. The main approaches that are used are the manual investigation and treating mutants not killed as equivalent. Based on the results, we recommend that further research on the equivalent mutant problem can develop a mature and useful auxiliary tool which can easily link to the existing mutation system. Such an extra tool assists people to solve the equivalent mutant problem when applying mutation testing more efficiently. Moreover, research on whether the equivalent mutant problem has a high impact on the accuracy of the mutation score is still needed, as the majority did not consider the equivalent mutant problem as a significant threat to the validation of the quality assurance processes. Also, 56.5% of the studies are lacking an explanation as to how they are dealing with overcoming the equivalent mutant problem; this again calls for more attention on reporting mutation testing *appropriately*.

RQ2.2.4: Which techniques are used to reduce the computational cost when applying mutation testing?

Observations.

Since mutation testing requires high computational demands, cost reduction is necessary for applying mutation testing, especially in an industrial environment. We summarized the use of such computational cost reduction techniques when using mutation testing in Table 2.15. Please note that we excluded the runtime optimisation and selective mutation techniques. We opted to exclude this because the runtime optimisation is related to tool implementation, which is not very likely to appear in the papers under our research scope, while the second one, selective mutation, is adopted by all the papers.

First of all, we noticed that 131 articles (68.6%) did not mention any reduction techniques, e.g., Außerlechner et al.[61] and Baker and Habli[62]. If we take into account those papers that used the runtime optimisation and selective mutation, one plausible explanation for the numerous "n/a" instances is a lack of awareness of *properly* reporting mutation testing, as we mentioned earlier. Secondly, random selection of the mu-

Table 2.15: Summary of cost reduction

Cost Reduction Technique	Number of papers
n/a	131
fixed number	28
weak mutation	15
mutant sample	11
selection strategy	8
higher-order	1

tants based on a fixed number comes next (28 instances, e.g., Namin and Andrews [262] and Staats et al. [327]), followed by weak mutation (15 instances, e.g. Hennessy and Power [176] and Vivanti et al. [353]) and mutant sampling (11 cases, e.g. Arcuri and Briand [54] and Stephan and Cordy [331]). However, why is the technique of using a “fixed number" of mutants more popular than the others? We speculate that this could be because choosing a certain number of mutants is more realistic in real software development: the total number of mutants generated by mutation tools is enormous; while, realistically, only a few faults are made by the programmer during implementation. By fixing the number of mutants, it becomes easier to control the mutation testing process. Instead, relying on the weak mutation condition would require additional implementation efforts to modify the tools. It is also important to note that the difference between the “fixed number" and “mutant sample" choice: while the first one implies a fixed number of mutants, the second one relies on a fixed sampling rate. Compared to using a fixed number, mutant sampling sometimes cannot achieve the goal of reducing the number of mutants efficiently. In particular, it is hard to set one sample ratio if the size of the subjects varies greatly. For example, consider the following situation: one subject has 100,000 mutants while the other has 100 mutants. When the sample ratio is set to 1%, the first subject still has 1000 mutants left, while the number of mutants for the second one is reduced to one.

We performed a further analysis of the mutation tools in Table 2.14. We find that most tools adopted some types of cost reduction techniques to overcome the high computational expense problem. For mutation testing tools for Java, bytecode translation is frequently adopted while Mutant Schemata Generation (MSG) is used in two tools, MuJava and JAVALANCHE. Another thing to highlight is that MiLu used a special test harness to reduce runtime [196]. This test harness is created containing all the test cases and settings for running each mutant. Therefore, only the test harness needs to be executed while each mutant runs as an internal function call during the testing process.

Selective mutation is also widely applied in almost all the existing mutation testing tools (as shown in Table 2.12). This brings us to another issue: *is the selected subset of mutation operators sufficient to represent the whole mutation operator set?* When adopting selective mutation, some configurations are based on prior empirical evidence, e.g., Offutt et al.'s five sufficient Fortran mutation operators [271], and Siami et al.'s 28 sufficient C mutation operators [322]. However, most of the articles are not supported by empirical or theoretical studies that show a certain subset of mutation operators can represent the whole mutation operator set. As far as we know, most studies on selective mutation are merely based on Fortran [258, 271, 367] and C [65, 220, 261, 322] programs. Thereby, we recommend more empirical studies on selective mutation in programming languages other than Fortran).

Compared to Jia and Harman's literature review [197], most of the cost reduction techniques they surveyed have been adopted in our collection. Runtime optimisation techniques which they summarised, e.g., interpreter-based technique [210], compiler-based approach [113] and bytecode translation [239] have been widely adopted in existing mutation testing tools. However, the articles we have reviewed did not apply Firm Mutation [192] and advanced platforms support for mutation testing, such as SIMD [215] and MIMD machines [276]. For Firm Mutation, there is no publicly available tool to support this method; thus, other researchers cannot adopt this approach conveniently. As for advanced platforms, these machines are not easy for other researchers in the mutation testing field to obtain. One exception is the cost reduction technique with a *fixed number*, which was not covered by Jia and Harman [197]. As mentioned earlier, the fixed-number-of-mutants technique is different from mutant sampling as the former selects a subset of mutants based on a fixed number rather than a ratio. We speculated the reason why Jia and Harm [197] did not include this method is that reduction based on a fixed number is too simple to be considered as a real technique for cost reduction in mutation testing. However, such a technique is surprisingly popular among the applications of mutation testing.

SUMMARY.

Based on the above discussion, we infer that the problem of the high computational cost of mutation testing can be adequately controlled using the state-of-art reduction techniques, especially selective mutation and runtime optimisation. Selective mutation is the most widely used method for reducing the high computational cost of mutation testing. However, in most cases, there are no existing studies to support the prerequisite that selecting a particular subset of mutation operators is sufficient to represent the whole mutation operator set for other programming languages instead of C and Fortran. Therefore, one recommendation is to conduct more empirical studies on selective mutation in various programming languages. Random selection of the mutants based on a fixed number (28 papers) is the most popular technique used to reduce the computational cost. The other popular techniques are weak mutation and mutant sampling. Besides, a high percentage of the papers (68.6%) did not report any reduction techniques used to cope with computational cost when applying mutation testing; this again should serve as a reminder for the research community to pay more attention to *properly* reporting mutation testing in testing experiments.

RQ2.2.5: WHAT SUBJECTS ARE BEING USED IN THE EXPERIMENTS (REGARDING PROGRAMMING LANGUAGE, SIZE, AND DATA AVAILABILITY)?

OBSERVATIONS.

To analyse the most common subjects used in the experiments, we focus on three attributes of the subject programs, i.e., programming language, size and data availability. We will discuss these three attributes one by one in the following paragraphs.

Table 2.16 shows the distribution of the programming languages. We can see that Java and C dominate the application domain (66.0%, 126 instances). While JavaScript is an extensively used language in the web application domain, we only found three research studies in our datasets that applied mutation testing for this programming language. The potential reasons for this uneven distribution are unbalanced data availability and the complex nature of building a mutation testing system. The first cause, uneven data availability, is likely instigated by the fact that existing, well-defined software repositories such as `SIR` [126], `SF100` [146] are based on C and Java. We have not encountered such repositories for JavaScript, C# or SQL. Furthermore, it is easier to design a mutation system targeting one programming language; this stands in contrast to many web applications, which are often composed out of a combination of JavaScript, HTML, CSS, etc; thus, this increases the difficulty of developing a mutation system for these combinations of programming languages. It is also worth noticing that we have not found any research on a purely functional programming language in our research scope.

When considering the size of the subject programs, in addition to our collection, we also summarise the data presented in Jia and Harman's survey [197] in Table 2.17. In Jia and Harman [197]'s survey, they summarised all the programs used in empirical studies related to mutation testing including the program size and the total number of uses. In Table 2.17, to summarise the data from Jia and Harman, we first categorise the programs into five classes (as shown in the first column) according to their size in LOC, and then add up the total number of usages (as shown in the third column). It is important to note that the program data in Jia and Harman's survey [197] is different from ours: they collected the programs from empirical studies which aimed to evaluate the effectiveness of mutation testing, e.g., Mathur and Wong compared data flow criteria with mutation testing [244]. These studies are not part of our research scope: we focus on the application perspective while these works target the development of mutation testing approaches. The purpose of the comparison of Jia and Harman's data is to investigate the difference between two perspectives of mutation testing, i.e., the *development* and the *application* perspective. Moreover, they listed all the possible programs used in empirical studies while we only collected the *maximum* size of the programs.

As for our collection, studies involving preliminary (<100 LOC), small (100~10K LOC) subjects or studies with no information about programs size ("n/a" instances in Table 2.17) represent 80.6%(154 instances) of papers in our collection. Among these "n/a" cases, some papers (e.g., Xie et al. [371]) did not provide any information about the subject size (LOC), and a few cases (e.g., Baudry et al [67]) report outdated links. This high percentage of preliminary, small and "n/a" subjects indicates that mutation testing is rarely applied to programs whose size is larger than 10K LOC. We did find that only 35 studies use medium size subjects, which corresponds to 29.9%[3] of papers. Compared to Jia and Harman's data [197], preliminary programs account for 63.4%[4] of the study set, which is much higher than in our study. There are two possible causes for this finding. The one is that we only consider the maximum size of the programs in each study, we could subtract many potential cases that used preliminary programs from our results. The other is that a considerable number of the empirical studies on mutation testing that Jia and Harman reviewed date back to the 1990s. These early-stage research works

[3]Notice that we did not consider the number of "n/a" value when calculating the percentage.$(191-74=117)$.

Table 2.16: Summary of programming languages

Programming language	Number of papers
Java	92
C	34
Lustre/Simulink	8
C#	6
Fortran	6
n/a	6
C++	5
SQL	5
Eiffel	3
Spreadsheet	3
AspectJ	3
JavaScript	3
Enterprise JavaBeans application	2
C/C++	2
Ada	1
Kermeta	1
Delphi	1
ISO C++ grammar	1
PLC	1
Sulu	1
XACML	1
XML	1
HLPSL	1
PHP	1
other specification languages	10

on mutation testing mainly involving preliminary subjects. Also, we witness an increasing trend of medium and large size subject systems being used in studies on mutation testing; this shows the full potential of mutation testing in large-scale applications.

With regard to data availability, we observe the following: 49.7% of the studies provide open access to their experiments. Some studies, such as Staats et al.[328], used close-sourced programs from industry. There are also a few cases for which the source links provided by the authors are not accessible anymore, e.g., Jolly et al.[199]. We also found that several cases used open-source software corpora, such as SF110, but they only used a sample of the corpus (e.g., Rojas et al. [305]) without providing sample information. The others did not provide information about sources in their paper, e.g., Kanewala and Bieman[206].

Together with 6 instances of "n/a" in Table 2.16 and 74 in Table 2.17 (including subjects which cannot be measured as LOC, e.g., spreadsheet applications), it is worth noticing that subject programs used in the experiment should be clearly specified. Also, basic

[4]Similarly, we removed the cases of "n/a" here when calculating the percentage.(402 − 28 = 374).

Table 2.17: Summary of subject size

Subject Size	**No. of papers in our collection**	**No. of programs in Jia and Harman [197]**
n/a	74	28
small (100~10K LOC)	70	128
medium (10K~1M LOC)	35	9
preliminary (<100 LOC)	10	237
large (>1M LOC)	2	0

Note: The results of our collection are based on the *maximum* size of the programs used in each study while Jia and Harman's is based on *all* the programs.

Table 2.18: Summary of data availability

Data Availability	**Number of papers**
no	96
yes	95

information on the programming language, size and subject should also be clearly specified in the articles to ensure replicability.

Summary.

For the subjects used in the experiments in our survey, we discussed three aspects: programming language, size of subject projects and data availability. For programming languages, Java and C are the most common programming languages used in the experiments when applying mutation testing. There is a clear challenge in creating more mutation testing tools for other programming languages, especially in the area of web applications and functional programming (see the recommendation labelled as **R7** in Section 2.4.4).

As for the maximum size of subject programs, small to medium scale projects (100~1M) are widely used when applying mutation testing. Together with two large-scale cases, we can see the full potential of mutation testing as a practical testing tool for large industrial systems. We recommend more research on large-scale systems to further explore scalability issues (see the recommendation labelled as **R8** in Section 2.4.4).

The third aspect we consider is data availability. Only 49.7% of the studies that we surveyed provide access to the subjects used; this again calls for more attention on reporting test experiments *appropriately*: the authors should explicitly specify the subject programs used in the experiment, covering at least the details of programming language, size, and source.

2.4.3. Summary of Research Questions

We now revisit the research questions and answer them in the light of our observations.

RQ2.1: How is mutation testing used in quality assurance processes?

Mutation testing is mainly used as a fault-based evaluation method (70.2%) in different quality assurance processes. It assesses the fault detection capability of various testing techniques through the mutation score or the number of killed mutants. Adopting mutation testing to improve other quality assurance processes as a guide was first proposed by DeMillo and Offutt [116] in 1991 when they used it to generate test data. As a “high end" test criterion, mutation testing started to gain popularity as a building block in different quality assurance processes, like test data generation (36 instances), test case prioritisation (6 cases) and test strategy evaluation (6 instances). However, using mutation testing as part of new test approaches raises a challenge in itself, namely how to efficiently evaluate mutation-based testing? Besides, we found one limitation related to the “guide" role of mutation testing: mutation testing usually serves as a *where*-to-check constraint rather than a *what*-to-check improvement. Another finding of the application of mutation testing is that it often targets unit-level testing (72.0%), with only a small number of studies featuring higher-level testing showing the overall benefit of mutation testing. As a result, we conclude that the current state of the application of mutation testing is still rather limited.

RQ2.2: How are empirical studies related to mutation testing designed and reported?

First of all, for the mutation testing tools and mutation operators used in literature, we found that 47.6% of the articles adopted existing (open-access) mutation testing tools, such as MuJava for Java and Proteum for C. In contrast, we did encounter a few cases (27 in total) where the authors implemented their own tools or seeded mutants by hand. Furthermore, to investigate the distribution of mutation operators in the studies, we created a generalised classification of mutation operators as shown in Listing 1. The results indicate that certain programming languages lack specific mutation operators, at least as far as the mutation tools that we have surveyed concern.

Moreover, when looking at the two most significant problems related to mutation testing, the main approaches to dealing with the equivalent mutant problem are (1) treating mutants not killed as equivalent and (2) not investigating the equivalent mutants at all. In terms of cost reduction techniques, we observed that the “fixed number of mutants” is the most popular technique, although we should mention that we did not focus on built-in reduction techniques.

The findings above suggest that the existing techniques designed to support the application of mutation testing are largely still under development: a mutation testing tool with a more complete set of mutation operators or a flexible mutation generation engine to which mutation operators can be added, is still needed [211, 279]. In the same vein, a more mature and efficient auxiliary tool for assisting in overcoming the equivalent mutant problem is needed. Furthermore, we have observed that we lack insight into the impact of the selective mutation on mutation testing; this suggests that a deeper understanding of mutation testing is required. For example, if we know what particular kinds of faults mutation is good at finding or how useful a certain type of mutant is when testing real software, we can then design the mutation operators accordingly, such as Just et al. [201].

Table 2.19: Summary of poorly-specified aspects in empirical studies

The poorly-specified aspects in reporting mutation testing	Number of papers
test level (see Section 2.4.1)	84
mutation tool source (see Section 2.4.2)	92
mutation operators (see Section 2.4.2)	72
equivalent mutant problem (see Section 2.4.2)	108
reduction problem (see Section 2.4.2)	131
subject program source (see Section 2.4.2)	96

Based on the distribution of subject programs used in testing experiments or case studies, Java and C are the most common programming languages used in the experiments. Also, small to medium scale projects (100~1M LOC) are the most common subjects employed in the literature.

From the statistics of the collection, we found that a considerable amount of papers did not provide a sufficiently clear or thorough specification when reporting mutation testing in their empirical studies. We summarised the poorly-specified aspects of mutation testing in Table 2.19. As a result, we call for more attention on reporting mutation testing appropriately. The authors should provide at least the following details in the articles: the mutation tool (preferably with a link to its source code), mutation operators used in experiments, how to deal with the equivalent mutant problem, how to cope with high computational cost and details of the subject program (see the ninth recommendation labelled as **R9** in Section 2.4.4).

2.4.4. Recommendation for Future Research

In this section, we will summarise the recommendations for future research based on the insights obtained for the two main research questions (see Sections 2.4.1 through 2.4.3). We propose nine recommendations for future research:

- **R1: Mutation testing cannot only be used as *where*-to-check constraints but also to suggest *what* to check to improve test code quality.**

  As shown in Table 2.4 in Section 2.4.1, when mutation testing serves as a "guide", mutants generated by the mutation testing system are mainly used to suggest the location to be checked, i.e., *where*-to-check constraints. For example, the location of mutants is used to assist the localisation of "unknown" faults in fault localisation. The mutation-based test data generation also used the position information to generate killable mutant conditions. However, mutation testing is not widely considered to be a benefit to improve test code quality by suggesting *what* to check, especially in the test oracle problem. The *what*-to-check direction can be one opportunity for future research in mutation testing as a "guide" role.

- **R2: For testing approaches that are guided by a mutation approach, more focus can be given to finding an appropriate way to evaluate mutation-based testing in an efficient manner.**

When looking at the evaluation types in Table 2.5 in Section 2.4.1, we observe that 75.4% of the mutation-based testing techniques still adopt mutation faults to assess their effectiveness. This raises the question of whether the conclusions might be biased. As such, we open the issue of finding an appropriate way to evaluate mutation-based testing efficiently.

- **R3: Study the higher-level application of mutation testing.**

  In Section 2.4.1 we observed that mutation testing seems to mainly target unit-level testing, accounting for 72.0% of the studies we surveyed. This reveals a potential gap in how mutation testing is currently applied. It is thus our recommendation that researchers pay more attention to higher-level testing, such as integration testing and system testing. The research community should not only investigate potential differences in applying mutation testing at the unit-level or at a higher level of testing but should also explore whether the conclusions based on unit-level mutation testing still apply to higher-level mutation testing. A pertinent question in this area could be, for example, whether an integration mutation fault can be considered as an alternative to a real bug at the integration level.

- **R4: The design of a more flexible mutation generation engine that allows for the easy addition of new mutation operators.**

  As shown in Table 2.8 in Section 2.4.1, 50.3% of the articles adopted the existing tools which are open-source, while we also found 27 instances of researchers implementing their own tool or seeding the mutants by hand. Furthermore, in Table 5.1 and Table 2.12, we can see certain existing mutation testing tools lack certain mutation operators. These findings imply that existing mutation testing tools cannot always satisfy all kinds of needs, and new types of mutation operators are also potentially needed. Since most existing mutation testing tools have been initialized for one particular language and a specific set of mutation operators, we see a clear need for a more flexible mutation generation engine to which new mutation operators can be easily added [279].

- **R5: A mature and efficient auxiliary tool to detect equivalent mutants that can be easily integrated with existing mutation tools.**

  In Section 2.4.2, the problem of equivalent mutants is mainly solved by manual analysis, assumptions (treating mutants not killed as either equivalent or non-equivalent) or no investigation at all during application. This observation leads to doubt about the efficacy of the state-of-art equivalent mutant detection. In the meanwhile, if there is a mature and efficient auxiliary tool which can easily link to the existing mutation system, the auxiliary tool can be a practical solution for the equivalent mutant problem when applying mutation testing. As a result, we call for a well-developed and easy to integrate an auxiliary tool for the equivalent mutant problem.

- **R6: More empirical studies on the selective mutation method can pay attention to programming languages other than Fortran and C.**

As mentioned in Section 2.4.2, selective mutation is used by all the studies in our research scope. However, the selection of a subset of mutation operators in most papers is not well supported by existing empirical studies, except for Fortran [258, 271, 367] and C [65, 220, 261, 322]. Selective mutation requires more empirical studies to explore whether a certain subset of mutation operators can be applied in different programming languages.

- **R7: More attention should be given to other programming languages, especially web applications and functional programming projects.**

  As discussed in RQ2.5 in Section 2.4.2, Java and C are the most common programming languages that we surveyed. While JavaScript and functional programming languages are scarce. JavaScript, as one of the most popular languages for developing web applications, calls for more attention from researchers. In the meanwhile, functional programming languages, such as Lisp and Haskell, are still playing an inevitable role in the implementation of programs; they thus deserve more focus in future studies.

- **R8: Application of mutation testing in large-scale systems to explore scalability issues.**

  From Table 2.17 in Section 2.4.2, we learn that the application of mutation testing to large-scale programs whose size is greater than 1M LOC rarely happens (only two cases). To effectively apply mutation testing in industry, the scalability issue of mutation testing requires more attention. We recommend future research to use mutation testing in more large-scale systems to explore scalability issues.

- **R9: Authors should provide at least the following details in the articles: mutation tool source, mutation operators used in experiments, how to deal with the equivalent mutant problem, how to cope with high computational cost and subject program source.**

  From Table 2.19 in Section 2.4.4, we remember that a considerable amount of papers inadequately reported on mutation testing. To help progress research in the area more quickly and to allow for replication studies, we all need to take care to be careful in how we report mutation testing in empirical research. We consider the above five elements to be essential when reporting mutation testing.

2.5. Threats to the Validity of this Review

We have presented our methodology for performing this systematic literature review and its findings in the previous sections. As conducting a literature review largely relies on manual work, there is the concern that different researchers might end up with slightly different results and conclusions. To eliminate this potential risk caused by researcher bias as much as possible, we follow the guidelines for performing systematic literature reviews by Kitchenman [212], Wohlin [365], and Brereton et al. [85]) whenever possible. In particular, we keep a detailed record of procedures made throughout the review process by documenting all the metadata from article selection to characterisation (see [23]).

In this section, we describe the main threats to the validity of this review and discuss how we attempted to mitigate the risks regarding four aspects: the article selection, the attribute framework, the article characterisation and the result interpretation.

2.5.1. Article Selection

Mutation testing is an active research field, and a plethora of realisations have been achieved as shown in Jia and Harman's thorough survey [197]. To address the main interest of our review, i.e., the actual application of mutation testing, we need to define inclusion/exclusion criteria to include papers of interest and exclude irrelevant ones. But this also introduces a potential threat to the validity of our study: unclear article selection criteria. To minimise the ambiguity caused by the selection strategies, we carried out a pilot run of the study selection process to validate our selection criteria among the three authors. This selection criteria validation led to a tiny revision. Besides, if there is any doubt about whether a paper belongs in our selected set, we had an internal discussion to see whether the paper should be included or not.

The venues listed in Table 2.2 were selected because we considered them to be the key venues in software engineering and most relevant to software testing, debugging, software quality and validation. This presumption might result in an incomplete paper collection. In order to mitigate this threat, we also adopted snowballing to extend our set of papers from pre-selected venues to reduce the possibility of missing papers. Moreover, we also ran two sanity checks (as mentioned in Section 2.3.2) to examine the completeness of our study collection, and recorded the dataset in each step for further validation.

Although we made efforts to minimise the risks with regard to article selection, we cannot make definitive claims about the completeness of this review. We have one major limitation related to the article selection: we only considered top conference or journal papers to ensure the high quality while we excluded article summaries, interviews, reviews, workshops (except the International Workshop on Mutation Analysis), panels and poster sessions. Vice versa, sometimes we were also confronted with a vague use of the "mutation testing" terminology, in particular, some papers used the term "mutation testing", while they are doing fault seeding, e.g., Lyu et al. [235]. The main difference between mutation testing and error seeding is the way how to introduce defects in the program [22]: mutation testing follows certain rules while error seeding adds the faults directly without any particular techniques.

2.5.2. Attribute Framework

We consider the attribute framework to be the most subjective step in our approach: the generalisation of the attribute framework could be influenced by the researcher's experience as well as the reading sequence of the papers. To generate a useful and reasonable attribute framework, we followed a two-step approach: (1) we first wrote down the facets of interest according to our research questions and then (2) derived corresponding attributes of interest. Moreover, for each attribute, we need to ensure all possible values of each attribute are available, as well as a precise definition of each value. In this manner, we can target and modify the unclear points in our framework quickly. In particular, we conducted a pilot run for specifically validating our attribute framework. The results led to several improvements to the attribute framework and demonstrated the applicability of the framework.

2.5.3. Article Characterisation

Thanks to the complete definitions of values for each attribute, we can assign the value(s) to articles in a systematic manner. However, applying the attribute framework to the research body is still a subjective process. To eliminate subtle differences caused by our interpretation, we make no further interpretation of the information extracted from the papers in the second pilot run of validation. In particular, if a detail is not specified in a paper, we mark it as "n/a". Furthermore, we listed our data extraction strategies about how to identify and classify the values of each attribute in Section 2.3.3.

2.5.4. Result Interpretation

Researcher bias could cause a potential threat to validity when it comes to the result interpretation, i.e., the author might seek what he expected for in the review. We reduce the bias by (1) selecting all possible papers in a manner that is fair and seen to be fair and (2) discuss our findings based on statistical data we collected from the article characterisation. Also, our results are discussed among all the authors to reach an agreement.

2.6. Conclusion

In this chapter, we have reported on a systematic literature review on the *application perspective* of mutation testing, clearly contrasting previous literature reviews that surveyed the main development of mutation testing, and that did not specifically go into how mutation testing is applied (e.g., [197, 241, 279]). We have characterised the studies that we have found on the basis of seven facets: (1) the role that mutation testing has in quality assurance processes; (2) the quality assurance processes (including categories, test level and testing strategies); (3) the mutation tools used in the experiments; (4) the mutation operators used in the experiments; (5) the description of the equivalent mutant problem; (6) the description of cost reduction techniques for mutation testing; and (7) the subject software systems involved in the experiments (in terms of programming language, size and data availability). These seven facets pertain to our two main research questions: **RQ1** *How is mutation testing used in quality assurance processes?* and **RQ2** *How are empirical studies related to mutation testing designed and reported?*

Figure 2.1 shows our main procedures to conduct this systematic literature review. To collect all the relevant papers under our research scope, we started with search queries in online libraries considering 17 venues. We selected the literature that focuses on the supporting role of mutation testing in quality assurance processes with sufficient evidence to suggest that mutation testing is used. After that, we performed a snowballing procedure to collect missing articles, thus resulting in a final selection of 191 papers from 22 venues. Through a detailed reading of this research body, we derived an attribute framework that was consequently used to characterise the studies in a structured manner. The resulting systematic literature review can be of benefit for researchers in the area of mutation testing. Specifically, we provide (1) guidelines on how to apply and subsequently report on mutation testing in testing experiments and (2) recommendations for future work.

The derived attribute framework is shown in Table 2.3. This attribute framework generalises and details the essential elements related to the *actual* application of mutation testing, such as in which circumstances mutation testing is used and which mutation testing tool is selected. In particular, a generic classification of mutation operators is constructed to study and compare the mutation operators used in the experiments described. This attribute framework can be used as a reference for researchers when describing mutation operators. We then presented the characterisation data of all the surveyed papers in our GitHub repository [23]. Based on our analysis of the results (in Section 2.4), four points are key to remember:

1. Most studies use mutation testing as an assessment tool; they target the unit level. Not only should we pay more attention to higher-level and specification mutation, but we should also study how mutation testing can be employed to improve the test code quality. Furthermore, we also encourage researchers to investigate and explore more interesting applications for mutation testing in the future by asking such questions as: what else can we mutate? (Sections 2.4.1—2.4.1)

2. Many of the supporting techniques for making mutation testing truly applicable are still under-developed. Also, existing mutation tools are not complete with regard to the mutation operators they offer. The two key problems, namely the equivalent mutant detection problem and the high computation cost of mutation testing issues, are not well-solved in the context of our research body (Sections 2.4.2—2.4.2).

3. A deeper understanding of mutation testing is required, such as what particular kinds of faults mutation testing is good at finding. This would help the community to develop new mutation operators as well as overcome some of the inherent challenges (Section 2.4.3).

4. The awareness of *appropriately* reporting mutation testing in testing experiments should be raised among the researchers (Section 2.4.3).

In summary, the work described in this chapter makes following contributions:

1. A systematic literature review of 191 studies that apply mutation testing in scientific experiments, which includes an in-depth analysis of how mutation testing is applied and reported on.

2. A detailed attribute framework that generalises and details the essential elements related to the *actual* use of mutation testing

3. A generic classification of mutation operators that can be used to compare different mutation testing tools.

4. An actual characterisation of all the selected papers based on the attribute framework.

5. A series of recommendations for future work including valuable suggestions on how to report mutation testing in testing experiments in an appropriate manner.

3
An Investigation of Compression Techniques to Speed up Mutation Testing

Mutation testing is widely considered as a high-end test coverage criterion due to the vast number of mutants it generates. Although many efforts have been made to reduce the computational cost of mutation testing, in practice, the scalability issue remains. In this chapter, we explore whether we can use compression techniques to improve the efficiency of strong mutation based on weak mutation information. Our investigation is centred around six mutation compression strategies that we have devised. More specifically, we adopt overlapped grouping *and* Formal Concept Analysis *(FCA) to cluster mutants and test cases based on the reachability (code coverage) and necessity (weak mutation) conditions. Moreover, we leverage mutation knowledge (mutation locations and mutation operator types) during compression. To evaluate our method, we conducted a study on 20 open source Java projects using manually written tests. We also compare our method with pure random sampling and weak mutation. The overall results show that mutant compression techniques are a better choice than random sampling and weak mutation in practice: they can effectively speed up strong mutation 6.3 to 94.3 times with an accuracy of >90%.*

This chapter has been published in the Proceedings of the 12th International Workshop on Mutation Analysis (Mutation 2017) [391] (2017 IEEE International Conference on Software Testing, Verification and Validation Workshops) and Proceedings of the 11th International Conference on Software Testing, Verification, and Validation (ICST) [392].

3.1. INTRODUCTION

Mutation testing has been actively investigated as a technique to evaluate the quality of test suites [197]. The main idea is (i) to introduce small syntactic changes (*mutants*) into the production code using *mutation operators*, and (ii) to measure the ability of a given test suite in detecting them [279]. One of the benefits reported in literature is that mutation testing provides a better measure of the fault detection capability of test suites compared to other test coverage criteria [144, 228, 244]. Despite its well-known advantages, mutation testing remains an extremely expensive activity since it requires to re-run the test suites against each mutant, whose number increases exponentially with the size of the program under test [93].

To address this limitation, several methods have been proposed and these can be classified in three main categories [277]: (*do fewer*) selecting fewer mutants to evaluate [41, 271], (*do smarter*) using run-time information to avoid unnecessary test executions [185, 200], (*do faster*) reducing the execution time for each single mutant [344]. Techniques falling into the first category are the most investigated. Indeed, researchers have proposed various strategies to sample mutants, such as random sampling [195], mutation operator selection [271], clustering [188], static analysis [218], and machine learning based sampling [333].

Recently, Gopinath et al. [160] have challenged the effectiveness and efficiency of mutation reduction strategies: their empirical evaluation with eight common mutant reduction techniques showed that *none* of them *provide any practical advantage over pure random sampling*. Although some techniques showed small improvements in effectiveness, the gains do not compensate for the extra overhead. Therefore, there is a need for reduction techniques that are not only more effective, but also more efficient compared to random sampling.

This chapter originates from the insights of Gopinath et al. [160] and focuses on the mutant reduction technique recently proposed by our preliminary work [391]. We originally tackle the problem of reducing the cost of mutation testing by combining *do fewer* and *do smarter* techniques through data compression methods. First, *weak mutation* (*do smarter*) is used to determine which mutants lead to an *infection state* through one single execution of the test suite against the original program. Then, *formal concept analysis* [364] (FCA) is applied to derive the *maximal groupings* [391], which are two-way clusters of mutants and tests. Each *maximal grouping* is composed of a set of mutants M and a set of tests T with the property that any mutant in M is weakly killed by any test in T. Finally, a *do fewer* strategy is applied by running one single test case (test selection) against one single mutant (mutant sampling) from each maximal grouping [391]. Our initial empirical study with five Java programs and automatically generated unit test suites showed that FCA reduces the execution time of mutation testing by up to 85%.

In this chapter, we spot two important limitations of FCA that can affect its ability to correctly estimate strong mutation. First, FCA groups mutants and tests according to weak mutation only: mutants leading to an infection state when running the same test case t (or set of tests) are assumed to be *redundant*. This is why only one mutant in each maximal grouping is evaluated for strong mutation. However, mutants that are redundant in terms of weak mutation are not necessarily redundant in terms of strong mutation, because, for example, they are injected in different code locations (e.g., different methods) or are generated by different mutation operators. Second, we previously [391] focused only on *maximal groupings*, which may leave some tests and/or some mutants not assigned to any maximal grouping. As consequence, the estimated mutation score may be inaccurate.

To overcome these limitations, we enhance FCA with (i) *mutant location* and (ii) *mutation operator type* information when grouping mutants and tests according to weak mutation. This prevents mutants infecting different statements or generated by different operators to be inserted in the same grouping. We also investigate maximal and non-maximal groupings to prevent final clusters from missing test cases and/or mutants.

To evaluate the benefits of our enhancements, we conducted an empirical study with 20 open-source Java projects and using the test suites manually written by the original developers. Then, we compare the different variants of the FCA-based technique with and without our enhancements and against weak mutation and pure random sampling (with 10% as sampling percentage).

Our results show that FCA with our enhancements is more accurate in estimating the (strong) mutation score compared to (i) the original FCA-based technique by our preliminary work [391], (ii) random sampling, and (iii) weak mutation. In particular, we find that the compression strategy based on non-maximal groupings and enriched with mutant location information (referred as *overlap+mloc* in the remainder of the chapter) estimates the strong mutation score with an average absolute error of 5% and an average accuracy of 93% while being five time faster than strong mutation, i.e., its average speed up is 5X. Instead, random sampling achieves a higher absolute error of 13% while requiring the same execution time of *overlap+mloc*, i.e., its average speed up is 5X as well. The other compression strategies lead to larger speed-up scores (up to 18X) but with the cost of having a larger absolute error, which ranges between 5% (i.e., the absolute error of *overlap+mloc*) and 13% (i.e., the absolute error of random sampling) on average. Therefore, our findings challenge prior results [160] as we find that mutation strategies based on compression methods (and FCA in particular) are more effective and/or more efficient than random sampling. Finally, weak mutation is the fastest technique but it also produces the largest absolute error of 23% on average.

As a final remark, we observe that all FCA-variants allow to estimate whether each individual mutant is strongly killed or not based on relatively few test executions by relying on the two-way clusters generated with FCA. Instead, random sampling does not take into account relationships between mutants and test cases, possibly also leading to underrepresented areas of production code in the estimation.

3.2. Background and Related Work

In this section, we begin with an overview of mutation reduction techniques in literature and the works that motivate our approach. Then, we introduce the crucial concepts and theories on which our approach is based.

3.2.1. Mutation Reduction Strategies

Techniques for reducing the high computational cost have been an active area of research. Offutt and Untch [277]'s literature review summarises these approaches into three categories: *do fewer*, *do smarter* and *do faster*. The most well-known techniques for reducing the computational cost of mutation testing are *random sampling* [41], *selective mutation* [271], *weak mutation* [185] and *mutant schema* [345]. The aforementioned methods are independent of the program under test which can be flexibly combined with our methodology.

More recently, researchers have aimed to make further gains by including run-time information; a widely-adopted strategy is to execute the test suite on the original program before mutation execution to avoid unnecessary executions. *Coverage-based optimisation* filters out test executions when a test case does not cover the mutated statement; this optimisation is in use in existing tools such as JAVALANCHE [313], Major [204] and PIT/PiTest [3]. *Infection-based optimisation* on the other hand only executes a test case on a mutant when the test infects the state of the mutant, filtering out weakly live mutants (whose execution states are different from the original code).

Just et al. [200] improved upon this by only executing a test on a mutant if the execution state of the mutated expression propagates to a top-level expression; they also partitioned mutants based on their intermediate results. Ma and Kim [236] applied a similar idea to cluster mutants for each test case by comparing the values of innermost expressions. Compared to Just et al. [200] and Ma and Kim [236]'s, we partition mutants for all test cases instead of targeting each test case.

Mutant clustering's aim is to reduce the number of mutants based on the similarity of mutants instead of random sampling. Hussain [188] applied clustering algorithms (e.g. K-means), however, the approach requires the execution of all mutants against all the test cases, which cannot reduce the overhead during the mutation execution. Later, Ji et al. [195] measured the similarity of the mutants using domain analysis. They divide mutants based on static control flow analysis. But they only manually analysed the clustering accuracy without indicating the runtime overhead caused by the domain analysis. Different from these works, our approach groups mutants based on their reachability and necessity conditions against the tests.

An approach that eliminates redundant mutants is *mutant subsumption* (e.g. [46, 202, 219]). However, mutant subsumption requires full knowledge of the mutation kill matrix, which requires the execution of every mutant against every test. Computationally this process is more costly than traditional strong mutation, thus cannot be used to speed-up mutation execution. Furthermore, *test prioritization and reduction* are also used to speed up mutation testing, e.g. Zhang et al. [380].

Moreover, Zhang et al. have recently proposed *Predictive Mutation Testing* (PMT) to predict mutation testing results without execution [379]. They extracted 12 features from the programs and constructed a classification model to predict whether a mutant is killed or surviving. Their experiment showed that PMT could improve the efficiency of mutation testing by up to 151.4 times with a small loss in accuracy. Despite high efficiency, their approach needs to collect a series of program features, which requires different tools to fulfil; this is a substantial burden for the common programmer. Unlike Zhang et al. [379], we do not require any additional program features; the weak mutation information needed for our mutant clustering and data compression can be collected by our tool during the initial execution against the original program.

Our approach is an extension of our preliminary work [391]. Despite the encouraging results, we have identified two important limitations of our initial methods as mentioned before (in Section 3.1), i.e., (1) weak mutation information is not enough; and (2) FCA could lead to missing mutants and/or tests. To address these limitations, we propose another compressing strategy, i.e., *overlapped grouping*, which is the simplest and strictest clustering method. Moreover, we take full advantage of mutation location and operator type knowledge when compressing.

3.2.2. Mutant Compression

We now describe the core concepts behind our approach: *weak mutation* and *FCA-based compression technique [391]*.

Weak Mutation. For a test case t to kill a mutant m which mutates the statement s of a program P, there are three conditions [278]: (i) *reachability*: the execution of t must cover s; (ii) *necessity*: the execution state of m is different from the execution state of s; (iii) *sufficiency*: the incorrect state of m must propagate to the output causing a failure in t.

Weak mutation uses the necessity condition, i.e., a mutant is *killed* if its execution leads to a state change. For example, the expression `c=a*b` and its mutated version `c=a/b` have different outcomes (i.e., the mutant is weakly killed) if `a≠1`, `a≠0` and `b≠1`. Differently from *strong* mutation, *weak* mutation scores can be computed with one single execution of each test by instrumenting the mutated locations [149].

FCA-based compression technique [391]. *Formal Concept Analysis* (FCA) was originally a data analysis method and has shown to be a powerful mathematical technique to convey and summarize large amounts of information [364]. It takes as input the *formal context* which is a structure $C = (O, A, I)$ where O is the set of *objects*, A is the set of *attributes* while $I \subseteq O \times A$ is a binary relation between O and A. Then, FCA produces the *concept lattice*, which is a collection of *formal concepts* in the data ordered by *sub-concept* relations, i.e., from super-concepts to sub-concepts. Each *formal concept* is composed of (i) a group of objects sharing the same attributes, and (ii) all attributes that apply to the objects in the concept [364].

In mutation testing context, the objects in O are the mutants, the attributes in A are the test cases, and I is the mutant-by-test infection matrix. Then, FCA derives formal concepts that represent groups of mutants that are weakly killed by the same subset of tests. In other words, the output of FCA can be viewed as two-way clustering since mutants and tests are grouped in concepts such that all mutants in the same concept c are weakly killed by all tests in c. Among these concepts, *FCA-based compression technique* only considers the maximal concepts that are directly connected to the exit point in the lattice hierarchy that are referred to as *maximal groupings*.

After obtaining the maximal groupings from the concept lattice, this approach first compresses the mutant-by-test infection matrix by condensing the rows, i.e., select one mutant from each maximal grouping. Then to further perform the compression on the columns, there are three approaches for *test case selection*: (i) *random*: randomly select one test from each maximal grouping; (ii) *Set cover based*: find a sufficient subset of test cases that weakly kill all possible mutants. i.e., at each stage, choose the test that weakly kills the largest number of *uncovered* maximal groupings; (iii) *Sorting-based*: select the test cases with the largest number of maximal groupings at each stage until all possible

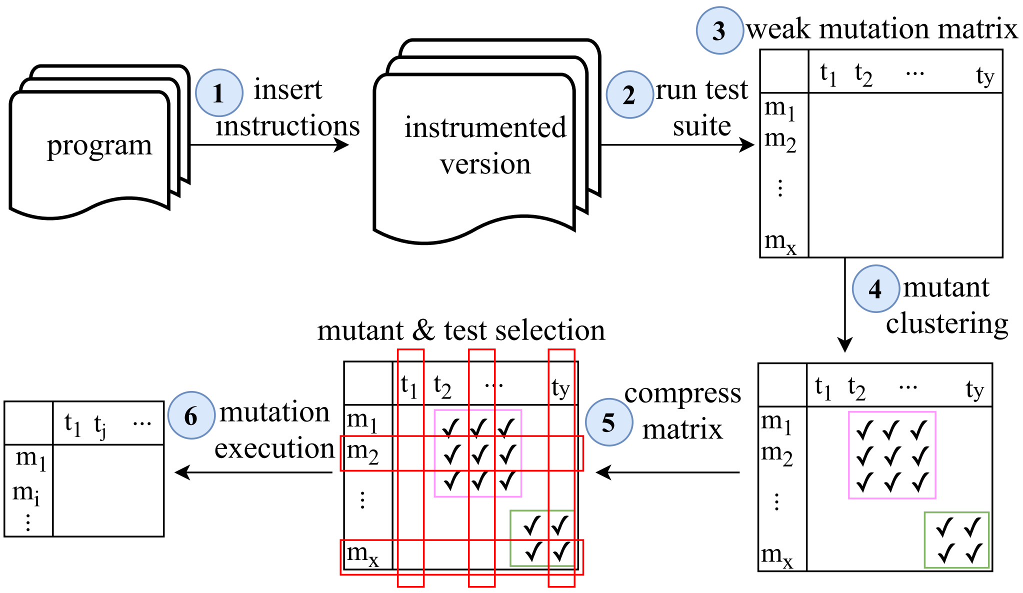


Figure 3.1: Overall methodology of mutation compression strategies

mutants are covered.

However, as our preliminary study [391] show, applying test case selection on FCA groupings leads to a relatively small (5.89%) reduction in execution time. Therefore, in this chapter, we do not use test case selection. Instead, we apply dynamic coverage-based optimisation [3, 204, 313] and infection-based optimisation [200, 236] (see Sec. 3.2) to filter out unnecessary executions.

3.3. Approach

3.3.1. Overall Methodology

Our 6-step compression strategy is illustrated in Figure 3.1:

(1) Instrumentation. We instrument the original program to keep track of the mutation locations: at every mutation point we insert all the mutants (mutated codes) right after the original one and assign a unique id to each mutant for later activation (we applied the technique of the *mutant schemata* [344]). To perform weak mutation, we also insert the comparison instructions at each mutation point to compare the intermediate states of the original program and mutated part (we compare the state after the first execution of the innermost expression that surrounds the mutant). We insert additional instructions to record information of each mutant including its location, operator type and mutation details (e.g., m_1 on Line 12 applies replace constant operator: $0 \to 1$).

(2) Test execution. Once instrumented, the test suite is executed once on the original program. During this stage, we record the mutants that are touched by the tests, as well as the ones which are weakly killed by the tests. Only the instructions related to weak mutation and mutant information collection are executed at this stage. No mutants are activated.

(3) Reachability and necessity analysis. The results of the previous stage are stored in the *mutant-by-test reachability* and *mutant-by-test necessity* matrices. Let P be the program under analysis and let T be the test suite; let $\mathbb{M}$ be the set of mutants for the program P generated by preselected mutation operators. A *mutant-by-test reachability* matrix is a $m \times n$ matrix where m is the number of mutants, n is the number of test cases in T, and an entry $x_{i,j}$ is a binary value indicating whether the statement containing the i-th mutant is executed ($x_{i,j} = 1$) or not ($x_{i,j} = 0$) by the j-th test $\in T$. The *mutant-by-test necessity* matrix has the same size as the *mutant-by-test reachability* matrix, but the binary entry $x_{i,j}$ represents the outcome of weak mutation ($x_{i,j} = 1$ indicates weakly killed).

(4) Mutant clustering. Using the two aforementioned matrices, we apply clustering to group similar mutants together. We consider two clustering methods: (1) the *overlapped* grouping, and (2) *FCA* grouping from our preliminary study [391] (See Section 3.2.2).

(5) Data compression. The mutant clusters are then used to compress the *mutant-by-test* matrix. The resulting matrix is likely to have lower dimensionality: the rows denote groups of mutants belonging to the same clusters; similarly, tests are grouped into clusters to form the columns. The compressed matrix is then used to apply mutants and execute tests for the strong mutation analysis. We take *mutation knowledge* into consideration during compression to achieve higher accuracy.

(6) Mutant Execution. The compressed matrix from the previous step is then used for the strong mutation analysis. Here, we load each mutant by its id and run the actual mutation execution against the tests.

The details of *overlapped grouping* and how we use *mutation knowledge* are described in the next sub-sections.

3.3.2. *overlapped* GROUPING

As mentioned earlier, the FCA-based compression technique could result in missing mutants and/or tests. Therefore, we propose the *overlapped* grouping to overcome this. The *overlapped* method is the simplest and strictest clustering method, i.e., elements are only grouped together if they are identical. Specifically, we first identify distinct mutants with regard to their reachability and necessity conditions against all the test cases. Subsequently, we group mutants having the same reachability and necessity conditions into one cluster.

The main difference between *overlapped* and *FCA* grouping is that *overlapped* grouping is stricter than *FCA* grouping. The *overlapped* grouping does not lose any information in the matrix, i.e., the clustering contains all the mutants. While *FCA* grouping loses some information since the main idea of *FCA* is to find the maximal sub-matrixes (or formal concepts), and if a mutant or a test does not belong to a sub-matrix, *FCA* removes this mutant or test.

In our example in Figure 3.2, there are three maximal groupings, which are $\{m5, m6|t1, t2\}$, $\{m4|t3\}$, and $\{m3|t4\}$. The other concepts in the lattice (e.g., $\{m2, m5, m6|t1\}$ in Figure 3.2) are already included in the maximal grouping by the sub-concept relation which is graphically represented by the hierarchy in the lattice. As for the *overlapped* grouping, it generates five clusters, i.e. $\{m1|t2\}$, $\{m2|t1\}$,$\{m3|t4\}$, $\{m4|t3\}$ and $\{m5, m6|t1, t2\}$. The example shows that for the clusters generated by the *overlapped* grouping no mutants are discarded, while the *FCA* grouping discards $m1$ and $m2$.

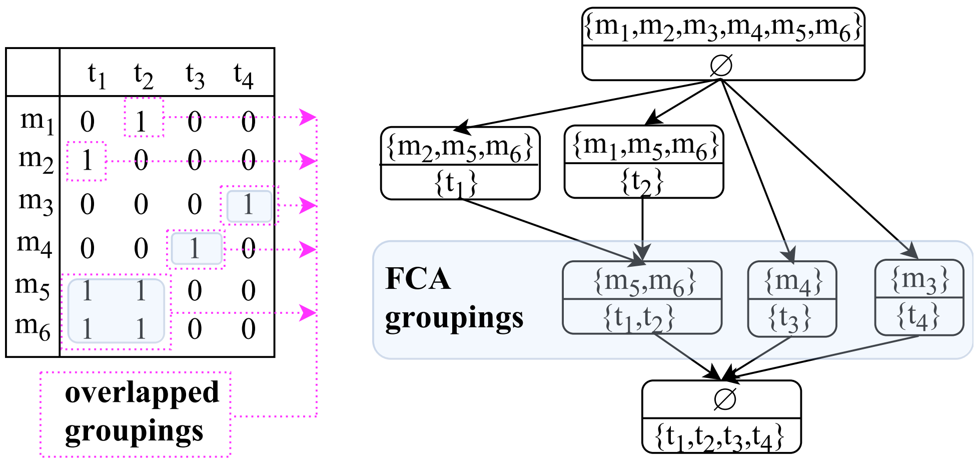


Figure 3.2: A toy program and its mutant clusters

3.3.3. MUTATION KNOWLEDGE

Once we have the mutant clusters, we compress the mutant-by-test infection matrix by condensing the rows, i.e., we select one/several mutants from each cluster to represent the whole group. To select the representative mutant(s), the FCA-based technique [391] adopted a random strategy which selects one mutant from each cluster at random, which we believe causes another limitation: based on weak mutation information alone, FCA could mis-cluster mutants at different code locations. Thus, we enhance FCA by adding *mutation knowledge*: (i) mutant location and (ii) mutation operator type.

More specifically, we investigate three mutant selection strategies: (1) random strategy; (2) random strategy with knowledge of the mutation operator type[1]; (3) random strategy with knowledge of the mutation location. The first one randomly chooses one mutant from each grouping/cluster as the representative mutant. The second strategy first divides each cluster into partitions by the type of mutation operator and then randomly selects one mutant from each partition. The third strategy partitions the cluster by the locations of the mutants (the line number) and then applies random selection; this guarantees that at least one mutant is selected for every potential mutation point. Notice that the second and third strategies might select more than one mutant from each grouping/cluster, which could lead to less speed-up in strong mutation.

To sum up, we devise one mutant clustering algorithm in addition to *FCA* as well as three mutant selection strategies, therefore, resulting in six compression strategies in total (as shown in Table 3.1).

3.4. EXPERIMENTAL STUDY

We conducted an empirical study to evaluate the effectiveness of the different compression strategies presented in the previous section. The *goal* of the study is to answer the following research questions:

- **RQ3.1:** *How accurate are different compression techniques?* We assess the ability

[1]Regarding the mutation operator type knowledge, we consider the operator type at a high level, e.g., arithmetic replacement operators.

Table 3.1: Summary of compression strategies

overlap	the combination of *overlapped* grouping and random strategy in mutant selection.
overlap+mop	the combination of *overlapped* grouping and random strategy with the knowledge of the mutation operator type in mutant selection.
overlap+mloc	the combination of *overlapped* grouping and random strategy with the knowledge of the mutation location in mutant selection.
fca	the combination of *FCA* grouping and random strategy in mutant selection.
fca+mop	the combination of *FCA* grouping and random strategy with the knowledge of the mutation operator type in mutant selection.
fca+mloc	the combination of *FCA* grouping and random strategy with the knowledge of the mutation location in mutant selection.

of the six compression strategies to estimate the strong mutation scores. We also asses their performance in comparison with random sampling and weak mutation.

- **RQ3.2:** *How do compression techniques perform in terms of speed-up?* We investigate the speed-up in terms of execution time that can be obtained when using each compression strategy over strong mutation. We also consider random sampling and weak mutation as baselines.

- **RQ3.3:** *What is the trade-off between accuracy and speed-up for the compression techniques?* We evaluate to what extent the compression strategies can reduce execution time while maintaining an accurate estimation of the strong mutation scores.

3.4.1. EXPERIMENTAL SETUP

To answer our research questions, we evaluated the six compression strategies using 20 open source projects publicly available on GitHub. Table 3.2 summarises the main characteristics of the selected projects. These projects have been randomly selected among the top 3000 GitHub repositories which (1) have most stars on 04/04/2017, (2) can be built using Maven, and (3) contain JUnit 4 test suites. In our study, we focus on the *manually-written* test suites available in the original project repositories.

As mentioned in Section 3.3, we first need one test execution against the original program to collect statement coverage (i.e., the *mutant-by-test reachability*) and the weak mutation information (i.e., the *mutant-by-test necessity* matrix). To collect weak and strong mutation information, we implemented our own prototype tool[2]. The instrumentation framework to generate mutants and detect the reachability and necessity condition is extracted from EvoSuite. Then, we integrated this instrumentation framework into our mutation testing runner. We record information about test cases (#id, method name, execution results, #touched mutants and #weakly killed mutants) and mutants

[2]All the tools, scripts and metadata for this experimental study are available in our GitHub repository [387].

(#id, mutation operator type, location and detailed information) for further analysis. After that, we run each test case against each mutant of the class under test (strong mutation) to establish the *mutant-by-test sufficiency* matrix which is used to evaluate our methods.

The mutation operators we adopted in this experiment are six method-level operators: `replace arithmetic`, `replace bitwise`, `replace comparison`, `replace variable`, `replace constant`, and `insert unary`. Further details about these mutation operators can be found in the paper by Fraser and Arcuri [149]. We opted for the mutation engine available in EvoSuite [149] because it instruments the production code at bytecode level and allows to directly measure the infection state for each mutant (weak mutation). To the best of our knowledge, no publicly-available mutation tool provides utilities for computing the weak mutation scores.

To answer the three **RQ**s, we selected another two mutation reduction techniques (see Section 3.2) for comparison: mutation sampling and weak mutation. We selected random sampling (*do fewer* strategy) as baseline because Gopinath et al. [160, 161] showed that none of the most common reduction strategies provide any practical advantage over *pure random sampling*. Moreover, we selected weak mutation (*do smarter* strategy) because it is one of the key components of all mutant compression techniques. Therefore, we considered it as an additional baseline to verify whether the other components of the compression strategies (e.g., computing the maximal groupings) are indeed needed. For random sampling, we set the sampling rate to 10% as suggested by Budd [93] and Acree [41]. They showed that 10% sampling could already estimate the mutation score with 99% of accuracy. It also corresponds to the sampling rate used by Gopinath et al. [160, 161]. Since random sampling and the mutant compression strategies involve random processing (i.e., in mutant selection), we carry out the corresponding random process 100 times for each project to address their randomised nature. In total, we compared eight mutation strategies: six compression strategies, random sampling (10%) and weak mutation.

3.4.2. Evaluation Metrics

To answer **RQ3.1**, we selected two well-known performance metrics: the *absolute error* and the *accuracy*. Let M be a given mutation strategy (e.g., random sampling); let $strong_M(C,T)$ be the percentage of mutants for a class C that are strongly killed by the test suite T; let $estimated_M(C,T)$ be the estimated percentage of mutants that are killed according to the strategy M; the *absolute error* is defined as follows:

$$AE(C,T) = \mid strong_M(C,T) - estimated_M(C,T) \mid \tag{3.1}$$

While the compression techniques select only a subset of the mutants for strong mutation, they can also estimate whether the non-selected mutants are killable by leveraging the groupings generated by FCA. Therefore, we use the accuracy as further performance metric, which is defined as follows:

$$accuracy(C,T) = (TP + TN)/total \tag{3.2}$$

where TP denotes the number of mutants that are strongly killed by T and that are also correctly identified by a given method M (*true positives*); TN is the number of mutants

Table 3.2: Subject Programs

PID	**Project**	**LOC**	**#Classes**	**#Tests**	**COV**	**#Mutants**			
						#Total	**#Covered**	**#Weakly Killed**	**#Strongly Killed**
1	assertj	24978	830	8545	0.90	59955	21677	19533	9178
2	checkstyle	31441	524	631	0.74	79464	18777	17738	9448
3	commons-lang	26578	264	2936	0.93	45630	43597	40453	33158
4	crawler4j	3745	57	11	0.27	3046	1017	853	505
5	dex-translator	4981	32	3	0.61	5812	1015	882	493
6	distributedlog	27976	697	339	0.43	21520	593	535	395
7	dynjs	34579	672	887	0.51	29148	14688	12960	8307
8	geotools	75236	991	670	0.38	62963	19852	17024	9135
9	graphhopper	26175	384	874	0.74	50319	11176	10168	7299
10	apns	1618	39	85	0.66	905	537	487	366
11	jctools	6262	133	43	0.82	7058	425	401	302
12	jfreechart	98334	657	2256	0.54	129417	14776	13057	5669
13	jpacman	1890	61	41	0.82	1606	1355	1159	580
14	junit-quickcheck	3038	67	354	0.98	1000	995	893	786
15	pac4j	5281	146	424	0.63	2703	1934	1689	929
16	pf4j	3021	67	24	0.28	1176	380	294	205
17	stream-lib	4767	77	121	0.83	11907	9920	9445	6924
18	telegrambots	1480	21	31	0.20	772	221	196	52
19	vraptor	12021	407	385	0.83	3490	2057	1652	1072
20	zt-zip	4255	84	3	0.71	2440	1049	906	422
Overall		397656	6210	2470	0.64	520331	166041	150325	95225

Note: Column "LOC" standing for the line of code is measured by sloccount[363]. Column "#Tests" is the total number of *passed* test cases under our ComMT tool. We marked the value with underline when the total number of passed test cases is less than the entire test suite size.The failures of the test cases in our tool is because the dependencies of these test cases need to be configured by Maven plugin; this cannot be solved in the current version of ComMT. The fifth column "COV" means the line coverage of the test suite which is measured by IntelliJ IDEA coverage runner.

that are not strongly killed by T and that are correctly identified by M (*true negatives*); *total* denotes the total number of mutants for the class C. To ease the comparison, we use the *mean* and *standard deviation* of *absolute error* and *accuracy* scores obtained for all classes in a given Java project.

For **RQ3.2** we consider the *speed-up* metric. When establishing the speed-up, we should first consider the overhead induced by an approach. For random sampling, we consider the overhead to be zero, as mutation sampling does not require any prerequisite knowledge. For weak mutation, the overhead consists of one single execution of the test suite against an instrumented version of the original program. For the compression strategies, the overhead is composed of the overhead incurred by both weak mutation and the compression procedure (mutant clustering and mutant selection):

$$overhead = exec\_time(weak\_mutation + compression) \tag{3.3}$$

The *speed-up* metric itself is computed using strong mutation with coverage-based optimisation as the baseline. We explicitly chose this optimisation as it is already integrated into several existing mutation testing tools (e.g., JAVALANCHE [313], Major [204] and PIT/PiTest [3]). The results of random sampling and compression strategies are the average values over 100 runs; for weak mutation, the execution time is zero. Then, the speed-up is defined as follows:

$$speed\text{-}up = \frac{exec\_time(strong\_mutation)}{exec\_time(M)} \tag{3.4}$$

where the denominator is the execution time of a method M computed as the sum of its overhead and the execution time needed to run the tests against the selected mutants.

For **RQ3.3**, we first provide a graphical comparison among the different mutation strategies by using the *speedup-error graphs*. In such a graph, the X-axis denotes the speed-up scores and the Y-axis shows the mean absolute error achieved by each strategy and for each project in our study; an "ideal" score would have a high X-value and a low Y-value. We also use the *speedup-accuracy graphs*, which plot the speed-up (mean) on the X-axis and the corresponding accuracy (mean) on the Y-axis; an "ideal" score would have a high X-value and a high Y-value. Although we may see some trends via graphical analysis, we would like to know which strategy achieves the best speed-up when accepting a given absolute error rate. Therefore, we consider the following absolute error thresholds: $\sigma_{e_1} = 5\%$, $\sigma_{e_2} = 10\%$, and $\sigma_{e_3} = 15\%$. Then, for each threshold σ_{e_i} and for each mutation strategy M, we count the number of projects for which M achieves the highest speed-up compared to the other strategies while yielding an absolute error score lower than σ_i. Similarly, we also consider three accuracy thresholds, i.e., $\sigma_{a_1} = 95\%$, $\sigma_{a_2} = 90\%$, and $\sigma_{a_3} = 85\%$. Different from the absolute error, we count the number of projects in which M achieves an accuracy higher than the threshold σ_{a_i}.

Statistical Analysis. To assess whether the differences among the various mutation strategies are statistically significant or not, we adopt Friedman's test [304] with $\alpha = 0.05$. It is a non-parametric test for comparing multiple treatments (mutation strategies) in the context of a multiple-problem analysis (i.e., multiple projects) [304]; it does not require data to be normally distributed and it is widely applied to compare randomised algorithms [154, 286] (e.g., random sampling). While Friedman's test reveals whether data distributions differ statistically, tests for pairwise comparison are needed to determine which treatment outperforms the others. For this, we use Conover's post-hoc procedure [104] and we further adjusted the obtained p-values using Holm-Bonferroni [181].

3.5. RESULTS

3.5.1. RQ3.1: ACCURACY

Table 3.3 reports the mean accuracy and absolute error scores for each project in our study as well as the corresponding standard deviation scores.

Absolute error. Focusing on the absolute error we observe that weak mutation performs worst with an error rate of 23% and a standard deviation of 22% on average. This

Table 3.3: Summary of Results for RQ3.1

	Absolute Error Summary (Mean / St. Dev.)								Accuracy Summary (Mean / St. Dev.)							
PID	Overlap			FCA			Random	Weak	Overlap			FCA			Random	Weak
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mutation	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mutation
1	0.09/0.17	0.08/0.16	**0.08/0.15**	0.13/0.17	0.11/0.16	0.11/0.16	0.13/0.17	0.20/0.27	0.90/0.17	0.91/0.17	**0.91/0.16**	0.83/0.20	0.86/0.18	0.90/0.17	-	0.76/0.29
2	0.06/0.11	0.06/0.10	**0.03/0.06**	0.08/0.12	0.07/0.10	0.03/0.07	0.10/0.13	0.13/0.17	0.93/0.12	0.93/0.11	**0.95/0.08**	0.91/0.14	0.92/0.12	0.95/0.08	-	0.86/0.17
3	0.07/0.11	0.04/0.07	**0.04/0.06**	0.13/0.13	0.08/0.08	0.06/0.07	0.14/0.16	0.18/0.19	0.88/0.13	0.91/0.10	**0.92/0.08**	0.74/0.19	0.80/0.16	0.90/0.09	-	0.81/0.18
4	0.10/0.12	0.06/0.07	**0.02/0.03**	0.11/0.12	0.07/0.08	0.03/0.03	0.07/0.09	0.28/0.18	0.89/0.13	0.91/0.10	**0.96/0.05**	0.87/0.13	0.89/0.11	0.96/0.05	-	0.71/0.17
5	0.13/0.16	0.09/0.16	**0.07**/0.16	0.14/0.17	0.10/0.16	0.07/0.16	0.10/0.16	0.15/**0.13**	0.86/0.16	0.87/0.16	0.90/0.16	0.85/0.17	0.87/0.16	**0.90**/0.16	-	0.85/**0.13**
6	0.03/0.10	0.03/0.08	**0.02/0.05**	0.06/0.12	0.05/0.09	0.04/0.08	0.14/0.21	0.11/0.17	0.97/0.10	0.97/0.09	**0.98/0.07**	0.92/0.14	0.95/0.11	0.96/0.09	-	0.82/0.21
7	0.12/0.15	0.09/0.13	**0.06/0.09**	0.16/0.16	0.11/0.13	0.07/0.09	0.19/0.19	0.33/0.27	0.86/0.16	0.88/0.14	**0.92/0.10**	0.80/0.18	0.84/0.15	0.91/0.11	-	0.67/0.27
8	0.05/0.09	0.03/0.06	**0.02/0.05**	0.07/0.11	0.05/0.08	0.03/0.05	0.12/0.16	0.24/0.26	0.94/0.10	0.95/0.08	**0.97/0.06**	0.91/0.13	0.93/0.10	0.96/0.06	-	0.75/0.26
9	0.11/0.13	0.07/0.09	**0.04/0.07**	0.15/0.14	0.09/0.10	0.05/0.08	0.12/0.13	0.19/0.18	0.85/0.15	0.88/0.12	**0.92/0.09**	0.76/0.16	0.81/0.14	0.90/0.09	-	0.80/0.17
10	0.06/0.10	0.04/0.05	**0.02**/0.05	0.09/0.12	0.06/0.07	0.03/**0.05**	0.09/0.12	0.25/0.28	0.92/0.12	0.95/0.07	0.97/0.05	0.89/0.13	0.92/0.09	**0.97/0.05**	-	0.75/0.27
11	0.23/0.23	0.22/0.22	**0.20**/0.22	0.23/0.23	0.22/0.22	0.20/0.21	0.29/**0.21**	0.38/0.33	0.77/0.23	0.77/0.22	**0.78**/0.22	0.77/0.23	0.77/0.22	0.78/**0.21**	-	0.62/0.33
12	0.05/0.07	0.04/0.05	**0.01/0.02**	0.09/0.09	0.05/0.06	0.02/0.02	0.03/0.03	0.24/0.23	0.91/0.09	0.92/0.07	**0.95**/0.05	0.87/0.11	0.89/0.10	0.95/**0.05**	-	0.76/0.23
13	0.15/0.16	0.10/0.11	**0.07/0.09**	0.16/0.17	0.11/0.12	0.07/0.09	0.11/0.12	0.39/0.25	0.83/0.17	0.85/0.14	0.90/**0.11**	0.81/0.18	0.84/0.15	**0.90**/0.11	-	0.61/0.25
14	0.04/0.07	0.03/0.06	**0.03/0.05**	0.10/0.11	0.07/0.08	0.07/0.08	0.16/0.18	0.09/0.12	0.96/0.08	0.96/0.07	**0.96/0.06**	0.78/0.18	0.82/0.15	0.95/0.06	-	0.87/0.13
15	0.10/0.15	0.07/0.13	**0.05/0.10**	0.17/0.15	0.12/0.13	0.11/0.11	0.19/0.18	0.24/0.26	0.88/0.15	0.91/0.13	**0.93/0.10**	0.71/0.19	0.80/0.17	0.89/0.12	-	0.74/0.25
16	0.08/0.13	0.06/0.09	**0.04/0.07**	0.11/0.14	0.09/0.11	0.06/0.07	0.21/0.19	0.21/0.24	0.90/0.14	0.93/0.09	**0.96/0.07**	0.86/0.15	0.89/0.12	0.95/0.08	-	0.77/0.23
17	0.10/0.13	0.07/0.10	**0.05/0.09**	0.15/0.14	0.09/0.10	0.05/0.09	0.12/0.14	0.22/0.20	0.84/0.15	0.87/0.13	**0.89/0.11**	0.76/0.18	0.79/0.15	0.88/0.11	-	0.76/0.22
18	0.10/0.12	0.07/0.10	0.03/0.04	0.11/0.14	0.08/0.11	**0.03/0.04**	0.12/0.14	0.30/0.21	0.89/0.12	0.91/0.11	0.94/0.09	0.87/0.15	0.89/0.13	**0.94/0.09**	-	0.70/0.21
19	0.06/0.11	0.04/0.08	**0.03/0.07**	0.11/0.15	0.07/0.11	0.06/0.10	0.17/0.20	0.20/0.25	0.94/0.11	0.95/0.09	**0.97/0.07**	0.88/0.17	0.91/0.13	0.96/0.08	-	0.77/0.24
20	0.14/0.17	0.09/0.12	**0.02**/0.03	0.15/0.17	0.09/0.12	0.02/**0.03**	0.07/0.10	0.31/0.29	0.85/0.17	0.86/0.15	**0.94**/0.08	0.85/0.17	0.86/0.15	0.94/**0.08**	-	0.66/0.28
Mean	0.09/0.13	0.07/0.10	**0.05/0.08**	0.13/0.14	0.09/0.11	0.06/0.08	0.13/0.15	0.23/0.22	0.89/0.14	0.90/0.12	**0.93/0.09**	0.83/0.16	0.86/0.14	0.92/0.10	-	0.75/0.22

Note: The text of "St. Dev." in first row stands for standard deviation. The second, third and fourth columns (i.e., Column "Simple", "w/ Mop" and "w/ Mloc") under Column "Overlap" correspond to method **overlap**, **overlap+mop** and **overlap+mloc** in Table 3.1. A similar rule applies to the fifth, sixth and seventh columns under Column "FCA". To be clear, we highlight the best (the lowest in the absolute value and the highest in the accuracy) scores in **bold**. Also, we underline the mean and std values of a compression technique if they are smaller (better) than the values of random sampling in the same project.

result is due to this strategy using all mutants that are weakly covered to approximate strong mutation coverage, yet, not all infected states propagate to changes observable in assertions.

Moreover, *overlap+mloc* produces the lowest mean absolute error in 19 out of 20 projects, followed by *fca+mloc* and *overlap+mop*. Overall, the strategies using *overlapped grouping* perform slightly better than FCA-based maximal groupings. To ease the comparison between the compression techniques and random sampling, in Table 3.3 we underline the mean and std values of a compression technique if they are smaller (better) than the values of random sampling in the same project. We can see that the six compression strategies outperform random *sampling* in terms of absolute error scores for most projects. The best (lowest) scores are obtained using the *overlap+mop*, *overlap+mloc*, *fca+mop* and *fca+mloc* approaches.

Looking at the six compression strategies, we can see that the mean absolute error and standard deviation scores are reduced when incorporating knowledge of mutation operators (*mop*) and location (*mloc*). For example, *overlap* achieves an error rate of 15% for the `jpacman` project, while when adding mutation operator and location knowledge it achieves an error rate of 10% and 7%, respectively. In general, we find that mutant location is more important than mutation operator, as the absolute error decreases by 4% on average, with a minimum improvement of 1% for `assertj` and a maximum improvement of 13% for `zt-zip`. This is an important novel contribution distinct from most related work, which consider mutation operators as a key element to detect redundant mutants or subsuming mutants (e.g. [46, 202, 219]). Hence, we found that *mutation location trumps mutation operator information when selecting mutants to evaluate for strong mutation.*

According to results of Friedman's test, the different mutation strategies achieve significantly different absolute error values (p-value = 10^{-16}). To better understand which strategies perform best, Table 3.4 reports the final ranking produced by Friedman's test as

Table 3.4: Ranking produced by Friedman's (larger rank indicates smaller error) and Statistical Significance by Conover's post-hoc procedure

ID	Algorithms	Rank	Significantly better than (PID)
(1)	overlap + Mloc	8.00	(2), (3), (4), (5), (6), (7), (8)
(2)	FCA + Mloc	6.55	(4), (5), (6), (7), (8)
(3)	overlap + Mop	6.30	(4), (5), (6), (7), (8)
(4)	FCA + Mop	4.55	(6), (7), (8)
(5)	Overlap	4.40	(6), (7), (8)
(6)	Random Sampling	2.70	(8)
(7)	FCA	2.50	(8)
(8)	Weak Mutation	1.10	-

well as the results of the pairwise comparison from Conover's procedure. As we can observe, *overlap* with mutant location knowledge (*mloc*) is ranked first and performs significantly better than all other strategies in the comparison. FCA based on maximal groupings with mutant location knowledge is ranked second and statistically outperforms all other strategies with lower ranks. Finally, random sampling is statistically worse than all *overlap* strategies and FCA enhanced with *mloc* and *mop*. Instead, the original FCA approach proposed by our preliminary study [391] is statistically equivalent to *random sampling* in terms of absolute error.

Accuracy. The results for the accuracy are also reported in Table 3.3. Since random sampling selects 10% mutants to evaluate in strong mutation, it cannot be used to estimate whether the other non-selected mutants are killable or not. Conversely, the six compression strategies can estimate whether each mutant is killable even if only a few mutants are actually evaluated for strong mutation. This is possible thanks to the two-way clusters generated by FCA: if a mutant is strongly killed, then we assume that all other mutants within its own cluster are killable as well. In weak mutation, we consider as strongly killable all mutants that lead to a state infection (i.e., the weakly killed ones).

Similar to results of the absolute error, the top three strategies are *overlap+mloc*, *fca+mloc* and *overlap+mop* in terms of accuracy. Again, weak mutation produces the worst accuracy in terms of both mean and standard deviation values. The three compression methods based on the overlapped grouping are slightly better than those in FCA. These differences are due to the fact that FCA considers only the maximal groupings as clusters from which selecting tests and mutants to run. However, as explained in Section 3.3, maximal groupings can miss some mutants, which therefore are not assigned to any cluster. Hence, we cannot accurately estimate whether the missed mutants are likely to be strongly killed or not based on the results of other selected mutants. We also notice that compression with additional mutant information can enhance the prediction accuracy. Finally, the finding that mutation location trumps mutation operator information still holds: the improvements range between 1% (for `commons-lang`) and 13% (for `junit-quickcheck`) in terms of accuracy.

Overlap with *mutation location* knowledge outperforms all other mutation strategies in terms of both **absolute error** and **prediction accuracy**. Random sampling is statistically worse than all mutant compression techniques.

Table 3.5: Summary of results for RQ3.2

	Overhead summary (compression overhead 10^{-4}%/ overall overhead%)						Speed-up summary (selected mutant% / speed-up)							
PID	Overlap			FCA			Overlap			FCA			Random	Weak
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mutation
1	1.04/0.06	**1.18/0.06**	1.04/0.06	1.05/0.06	0.21/0.06	0.78/0.06	8.4/52.1	9.9/27.3	12.5/22.4	3.1/**94.3**	4.9/43.3	9.0/25.8	10.0/22.7	0/**1747.8**
2	0.28/0.08	**0.30/0.08**	0.28/0.08	0.30/0.08	0.09/0.08	0.01/0.08	1.2/32.8	2.2/27.8	12.0/7.6	0.6/**42.1**	1.0/34.1	3.9/7.6	10.0/7.6	0/**1257.0**
3	2.92/0.27	3.19/0.27	3.01/0.27	**3.35/0.27**	0.17/0.27	0.02/0.27	15.0/8.8	22.1/5.4	30.7/4.8	3.2/**30.8**	8.5/14.2	25.6/5.4	10.1/4.9	0/**369.2**
4	0.01/0.30	0.02/0.30	**0.03/0.30**	0.02/0.30	0.01/0.30	0.02/0.30	2.0/27.3	5.1/13.0	17.4/3.8	1.3/**34.2**	3.0/14.5	11.3/3.8	10.0/3.8	0/**334.6**
5	0.01/1.10	0.02/1.10	**0.07/1.10**	0.03/1.10	0.01/1.10	0.03/1.10	0.6/37.6	1.6/21.0	16.0/3.9	0.4/**42.5**	1.1/22.0	5.8/3.9	10.0/4.0	0/**91.2**
6	0.02/9.23	0.02/9.23	0.03/9.23	**0.03/9.23**	0.02/9.23	0.01/9.23	0.7/5.5	1.1/4.9	2.0/4.6	0.4/6.3	0.7/5.4	1.1/4.9	10.0/**8.0**	0/**10.8**
7	0.03/0.03	0.04/0.03	0.04/0.03	**0.05/0.03**	0.02/0.03	0.01/0.03	9.7/11.2	14.5/8.6	25.2/3.1	3.5/**16.1**	6.9/11.5	18.8/3.2	10.0/3.1	0/**3280.9**
8	1.56/0.23	**2.09/0.23**	1.97/0.23	1.93/0.23	0.41/0.23	0.18/0.23	5.2/7.0	8.2/5.2	21.5/3.0	1.7/**13.2**	3.4/8.3	11.4/3.1	10.0/3.1	0/**432.9**
9	0.11/0.18	**0.15/0.18**	0.13/0.18	0.13/0.18	0.02/0.18	0.01/0.18	2.7/7.8	4.5/5.7	8.6/4.2	0.7/**36.9**	1.9/14.4	6.2/4.9	10.0/4.3	0/**561.7**
10	0.01/1.08	0.02/1.08	0.02/1.08	**0.05/1.08**	0.02/1.08	0.01/1.08	4.5/10.1	8.2/6.4	14.2/3.9	2.0/**15.0**	4.3/8.2	9.4/4.1	10.0/4.1	0/**92.9**
11	0.03/0.32	0.08/0.32	0.08/0.32	**0.12/0.32**	0.08/0.32	0.07/0.32	0.6/19.8	1.5/7.6	3.5/4.8	0.5/**19.8**	1.0/7.6	1.8/4.8	10.0/4.9	0/**310.9**
12	3.95/0.21	**4.43/0.21**	3.86/0.21	3.78/0.21	0.16/0.21	0.03/0.21	1.0/12.0	1.6/7.4	7.4/3.9	0.2/**64.9**	0.5/27.9	4.1/4.0	10.0/4.0	0/**481.9**
13	0.01/0.16	0.02/0.16	0.02/0.16	**0.03/0.16**	0.02/0.16	0.01/0.16	6.5/17.8	12.8/8.3	25.4/4.6	3.9/**21.9**	8.4/9.9	20.6/4.7	10.0/4.7	0/**630.9**
14	0.03/0.91	0.07/0.91	0.07/0.91	**0.11/0.91**	0.03/0.91	0.03/0.91	41.1/5.6	48.2/4.6	52.9/4.2	13.3/**11.1**	24.5/7.2	41.5/4.4	11.4/4.4	0/**110.2**
15	0.15/1.29	0.37/1.29	0.37/1.29	**0.45/1.29**	0.18/1.29	0.12/1.29	21.0/4.8	25.5/3.9	35.4/3.0	6.2/**9.1**	11.1/5.9	26.3/3.1	10.0/3.2	0/**77.5**
16	0.39/3.01	0.83/3.01	1.00/3.01	**1.58/3.01**	1.22/3.01	0.84/3.01	9.9/4.6	14.5/3.5	31.3/2.1	4.9/**6.8**	8.2/4.1	17.7/2.2	10.0/2.3	0/**33.3**
17	0.00/0.16	0.01/0.16	0.01/0.16	**0.01/0.16**	0.00/0.16	0.00/0.16	4.9/23.9	9.8/10.7	18.7/6.3	1.3/**94.6**	4.1/31.0	15.2/6.4	10.0/6.3	0/**626.7**
18	0.00/1.56	0.01/1.56	0.01/1.56	**0.01/1.56**	0.00/1.56	0.00/1.56	2.5/15.8	5.2/9.1	13.6/3.9	1.3/**18.1**	2.7/10.4	8.5/3.9	10.0/4.2	0/**63.9**
19	2.55/2.48	6.57/2.48	6.38/2.48	**8.59/2.48**	4.66/2.48	3.19/2.48	20.7/5.1	29.5/4.0	44.7/3.1	9.4/**8.7**	14.6/5.6	29.5/3.1	10.8/3.3	0/**40.3**
20	0.01/0.09	0.04/0.09	0.07/0.09	**0.09/0.09**	0.04/0.09	0.02/0.09	1.1/68.5	3.4/24.4	18.6/5.4	0.9/**69.0**	2.1/25.7	10.2/5.4	10.0/5.5	0/**1157.8**
Mean	0.66/1.14	0.97/1.14	0.92/1.14	1.09/1.14	0.37/1.14	0.27/1.14	7.97/18.91	11.47/10.44	20.58/5.13	2.94/**32.77**	5.65/**15.56**	13.90/5.44	10.12/5.42	0/**585.62**

Note: The second, third and fourth columns (i.e., Column "Simple", "w/ Mop" and "w/ Mloc") under Column "Overlap" correspond to method **overlap**, **overlap+mop** and **overlap+mloc** in Table 3.1. A similar rule applies to the fifth, sixth and seventh columns under Column "FCA". To be clear, we highlight the highest values in the overhead ratios and two strategies achieving the best speed-up scores in **bold**. Also, we underline the compression strategies that achieve better speed-up scores than random sampling in each project.

3.5.2. RQ3.2: SPEED-UP

Speed-up performance. Table 3.5 summarises the overall speed-up for the eight approaches in our comparison. For each project, we highlight the two strategies achieving the best speed-up scores in **bold**. Notice that speed-up measures the overall execution time of strong mutation divided by the overall execution time of a mutation strategy. Hence, higher values denote a larger improvement in execution time.

We observe that weak mutation shows the highest speed-up scores since it requires only one test suite execution against the original program. Except for weak mutation, FCA achieves the highest speed-up scores in 19 out of 20 cases. *FCA* is also faster than *random sampling*, which selects 10% mutants for strong mutation. Indeed, the former is 6.6 times faster than the latter on average, with a minimum speed-up of 2.6X (in `vraptor`) and a maximum one of 15.4X (in `stream-lib`). This is because *FCA* suggests on average less than 10% of mutants (with a minimum of 0.4% of mutants) to evaluate in strong mutation analysis. Instead, the sampling strategy constantly (and randomly) selects 10% of mutants to execute.

The only exception to the previous finding is represented by `distributedlog` for which *random sampling* is faster than *FCA*. In this case, the total percentage of mutants that are injected into statements covered by the test suite (*reachability condition*) is fairly low, being 2.8%. Thus, random sampling can achieve a considerable speed-up if we leverage the coverage-based optimisation, i.e., if we skip uncovered mutants (i.e., mutants of uncovered statements). Instead, *FCA* selects almost twice as many mutants for these projects.

To further ease the comparison, in Table 3.5 we underline the compression strategies that achieve better speed-up scores than random sampling in each project. We observe that *overlap*, *FCA+mop* and *overlap+mop* outperform random sampling in terms of speed-up for 19 projects out of 20. On average, they are respectively 3.7X, 3.0X, and 2.1X

Table 3.6: Ranking produced by Friedman's (smaller rank indicates better speed-up) and statistical significance by Conover's post-hoc procedure

ID	Algorithms	Rank	Significantly better than (PID)
(1)	Weak Mutation	1.00	(2), (3), (4), (5), (6), (7), (8)
(2)	FCA	2.10	(3), (4), (5), (6), (7), (8)
(3)	Overlap	3.50	(5), (6), (7), (8)
(4)	FCA + Mop	3.55	(6), (7), (8)
(5)	overlap + Mop	5.05	(6), (7), (8)
(6)	Random Sampling	6.25	(8)
(7)	FCA + Mloc	6.65	(8)
(8)	overlap + Mloc	7.90	-

faster than random sampling. It is worth noticing that the number of selected mutants does not directly determine the overall speed-up. For example, for the project `pac4j`, *overlap* selected 21.0% of mutants, which is larger than the percentage of mutants selected by mutation sampling (i.e., 10%). However, *overlap* achieves a larger speed-up of 4.8X against 3.1X of mutation sampling. The reason is that compression strategies uses weak mutation information to further filter out the unnecessary test executions, while mutation sampling does not.

From the comparison of the six compression strategies, we observe that including mutant location leads to selecting more mutants for strong mutation, thus, reducing the overall speed-up. For example, *overlap+mloc* achieves lower speed-up scores than *overlap* in all 20 projects. Moreover, by comparing the two strategies based on mutation location knowledge (i.e., *overlap+mloc*, and *fca+mloc*) with random sampling, we observe that the differences in terms of speed-up are small. Indeed, the average speed-up scores of *overlap+mloc*, *fca+mloc* and *random sampling* are 5.13, 5.44 and 5.42, respectively. Instead, selecting mutants according to mutation operator generates a lower number of mutants to evaluate in strong mutation compared to mutation location.

Our findings are confirmed by Friedman's test (shown in Table 3.6): the mutation strategies are statistically different in terms of speed-up scores (p-value = 10^{-16}). According to Conover's procedure, *weak mutation* and FCA statistically outperform all other mutation strategies. Moreover, random sampling is ranked sixth and is statistically more efficient than *overlap+mloc* only, although the difference is marginal as suggested by the average scores reported in Table 3.5. Instead, FCA, *overlap*, *FCA+mop* and *overlap+mop* are statistically superior to random sampling.

Overhead. In the previous paragraphs, we observed that most of mutation compression strategies are more efficient than random sampling. Here, we investigate the overhead that is due to the different steps that such strategies implement. In the following, we consider as *compression overhead* the execution time needed to compute the maximal and/or overlapped groupings; while the *overall overhead* is the sum of the *compression overhead*, the time for running all tests once for weak mutation, and the time to select the mutants.

Table 3.5 reports the *compression overhead* and *overall overhead* as a percentage (ratio) of the full execution time of each strategy, which also includes the time needed to run the selected tests and mutants for strong mutation. The highest values for each project are highlighted in **bold** face. From Table 3.5, we can observe that the compression overhead takes up less than 0.001% of the total execution time; thus, it is negligible with respect to the execution time of evaluating the selected mutants for strong mutation. The overall overhead accounts for up to 9.30% of the total execution time and weak mutation represents the larger portion of this overhead. Among the 20 projects, the overall overhead of the strategy *FCA* is likely higher than the other compression strategies. However, the differences among them are lower than 0.1%.

Weak mutation scores best among the eight techniques in terms of speed-up. Without considering weak mutation, four mutant compression strategies are statistically more efficient (have better speed-up scores) then random sampling.

3.5.3. RQ3.3: TRADE-OFFS

From the results of **RQ3.1** and **RQ3.2**, it is clear that the mutation strategies that perform best in terms of accuracy are also the more expensive to perform. Weak mutation and *FCA* grouping strategies perform best in terms of speed-up, while the *overlapped* grouping strategies, and *overlap+mloc* in particular, better approximate the strong mutation score. Therefore, in this section, we analyse the trade-offs between speed-up, absolute error and accuracy.

From Figure 3.3, we observe that weak mutation achieves the best speed-up, while its absolute error and accuracy typically score worst when compared to other techniques. FCA comes second in terms of the overall speed-up, but its absolute error as well as accuracy are better than weak mutation for most of the projects. We notice that *overlap* is slightly slower than FCA, but shows a small improvement in both absolute error and accuracy when compared to FCA. *FCA+mop* and *overlap+mop* have quite similar trade-offs considering speed-up and absolute error as their data points are very close to each other. However, in terms of speed-up and accuracy, *fca+mop* is slightly faster than *overlap+mop*, while *overlap+mop* is more accurate than *fca+mop*. Moreover, *overlap+mloc*, *fca+mloc* and *random sampling* have the same speed-up score; however, the absolute error of *random sampling* is higher than for the other two strategies. *Overlap+mloc* and *fca+mloc* are the most accurate strategies in terms of both absolute error and accuracy, but their speed-up performance is the least good.

Table 3.7 shows the number of projects for which each strategy M provide the best speed-up score at different thresholds of absolute error (σ_{e_i}) or accuracy (σ_{a_i}). As we can observe, *overlap+mloc* has the highest speed-up score when considering an absolute error ≤5% for 10 projects out of 20. This indicates that using mutation location leads to more accurate estimations of the actual strong mutation. When considering a 10% error rate threshold, *FCA+mop* and *FCA* show the largest speed-up for the majority of projects. Instead, when the goal is to reach an absolute error ≤15%, FCA has the best speed-up scores for 11 out of 20 projects. Moreover, we notice that random sampling performs worst as it has speed-up scores that are always lower then the other mutation strategies at the same (or higher) level of absolute error.

Similar results can be observed when considering different thresholds for accuracy. As reported in Table 3.7, *overlap+mloc* and *fca+mloc* show the best speed-up for accuracy ≥95% for the relative majority of the projects. When considering an accuracy ≥90%, *overlap* and *fca+mloc* show the best speed-up for 5 out of 20 projects each. Instead, if

Table 3.7: Number of projects for which each strategy M_j provides the best speed-up score at different thresholds

(a) Error Rate

Absolute Error	Overlap			FCA			Random	Weak
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mut.
≤5%	2	4	**10**	0	1	3	0	0
≤10%	4	1	1	5	**7**	1	0	1
≤15%	1	0	1	**11**	3	0	0	4
overall	7	5	12	**16**	12	4	0	5

(b) Accuracy

Accuracy	Overlap			FCA			Random	Weak
	Simple	w/ Mop	w/ Mloc	Simple	w/ Mop	w/ Mloc	Sampling	Mut.
≥95%	2	2	**9**	0	0	7	-	0
≥90%	**5**	4	2	3	1	**5**	-	0
≥85%	4	2	1	**8**	1	1	-	3
overall	11	8	12	11	2	**13**	-	3

Note: The second, third and fourth columns (i.e., Column "Simple", "w/ Mop" and "w/ Mloc") under Column "Overlap" correspond to method **overlap**, **overlap+mop** and **overlap+mloc** in Table 3.1. A similar rule applies to the fifth, sixth and seventh columns under Column "FCA". The last column "Weak Mut." stands for the strategy of weak mutation.

we focus on 85% of accuracy, we observe that FCA is the best approach to choose as it shows a speed-up ranging from 6.3X up to 94.3X compared to strong mutation. Finally, if we focus on 85% accuracy, we observe that FCA is the best approach to choose.

Overlap+mloc provides the best speed-up scores when the goal is to achieve an accuracy >95% or an absolute error <0.05. Other mutation compression strategies provide larger speed-up, but with a corresponding decrease in accuracy. Random sampling is less accurate and/or slower than all compression strategies.

3.5.4. Discussion

Looking at all the results, we can observe that random sampling with 10% sampling ratio is able to speed up (strong) mutation testing from 2.0 to 22.7X with an absolute error within 15% for 80% of the projects. Mutation sampling is also easy to apply in mutation tools as it does not require any prerequisite knowledge of the program context and mutation operators. However, *mutation strategies based on compression techniques achieve better speed-up (i.e., are more efficient) and/or lower absolute error than random sampling*. For example, *overlap+mloc* yields an absolute error which is always lower than 9% with a speed-up ranging between 2.0 to 53X. This represents an important finding if we consider the recent study by Gopinath et al. [160], which showed that other mutation reduction techniques (including *e-selective*) provide small or negligible improvements in effectiveness and are more expensive compared to random sampling.

Another disadvantage of random sampling is that it can estimate the overall mutation score, but cannot estimate whether each mutant is strongly killable or not (it only does for the sampled mutants). The overall mutation score is of course very important when assessing the test suite quality at a high level; however, Coles [100, slide 57] observed that programmers prefer to obtain specific insights into which mutants their test suite is able to kill. From this perspective, mutation compression strategies select a subset of “representative" mutants for the programmers to investigate. In addition, *overlap+mloc* and *FCA+mloc* guarantee that for every possible statement that can be mutated, at least one mutant will be selected. Although this may negatively affect the speed-up compared to random sampling, programmers can benefit from the killable mutant results at every possible mutant location.

3.6. Threats to Validity

Threats to external validity: Our results are based on mutants generated by the operators implemented in EvoSuite; these results might be different when using other mutation tools [220]. With regard to the subject selection, we chose 18 out of the 20 projects from GitHub’s top starred 3000 repositories; the selected projects differ in size, number of test cases and application domain.

Threats to internal validity: The main threat for our study is the implementation of the compression strategies. For FCA, we use its implementation available in MATLAB [245], which is a well-known scientific software. For the instrumentation and the mutation operators, we relied on their implementation available in EvoSuite [147]. Moreover, we carefully reviewed and tested all code for our study to eliminate potential faults in our implementation.

Threats to construct validity: The main threat is the measurement we used to evaluate our methods. We minimise this risk by adopting evaluation metrics that are widely used in research, as well as proper statistical analysis to assess the significance.

3.7. Conclusions

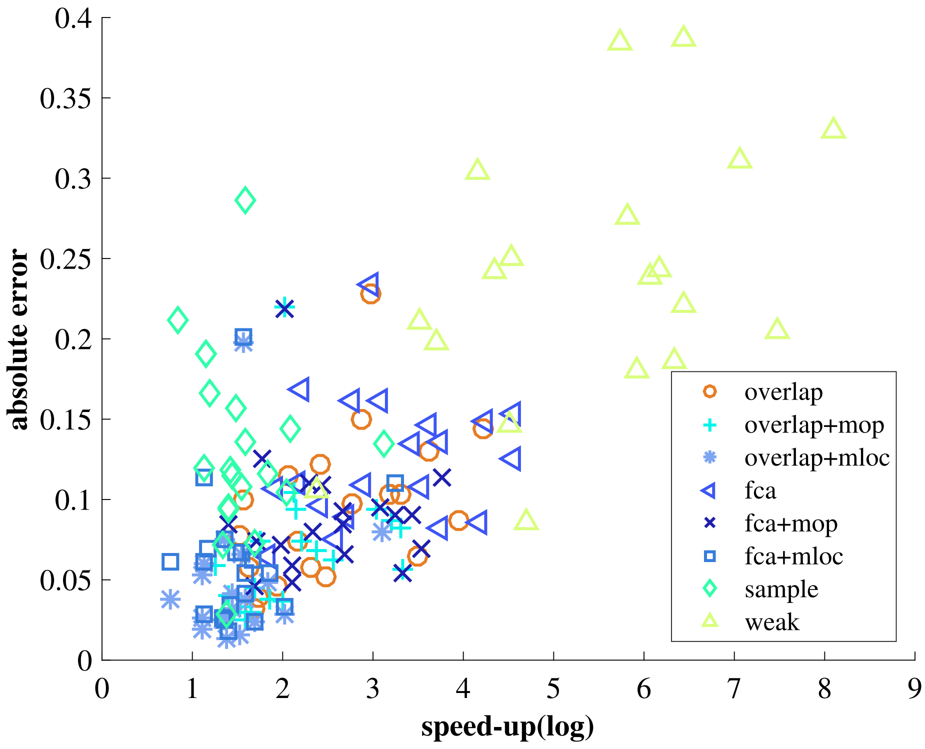
In this chapter, we have conducted a detailed investigation of different compression techniques to speed up mutation testing based on the work by our preliminary work [391]. We have enhanced our original *FCA-based compression strategy* in two distinct ways: (1) by proposing a novel mutant clustering algorithm, *overlapped grouping*, in addition to *FCA*; (2) by incorporating mutation location and mutation operator information in the compression procedure. Thereby, we have introduced and investigated six compression strategies based on two clustering algorithms and three mutant selection strategies.

The results of an empirical study with 20 open-source projects show that mutant compression techniques can effectively speed up strong mutation testing up to 94.3 times with an accuracy > 90%. FCA is the fastest strategy while $overlap + mloc$ is the most accurate. In comparison, *weak mutation* attains a higher absolute error (23%) and lower accuracy (75%). *Random sampling* with 10% as sampling percentage is statistically less accurate than all mutant compression strategies, and worse in terms of speed-up than four compression strategies (excluding the two with knowledge of mutation locations).

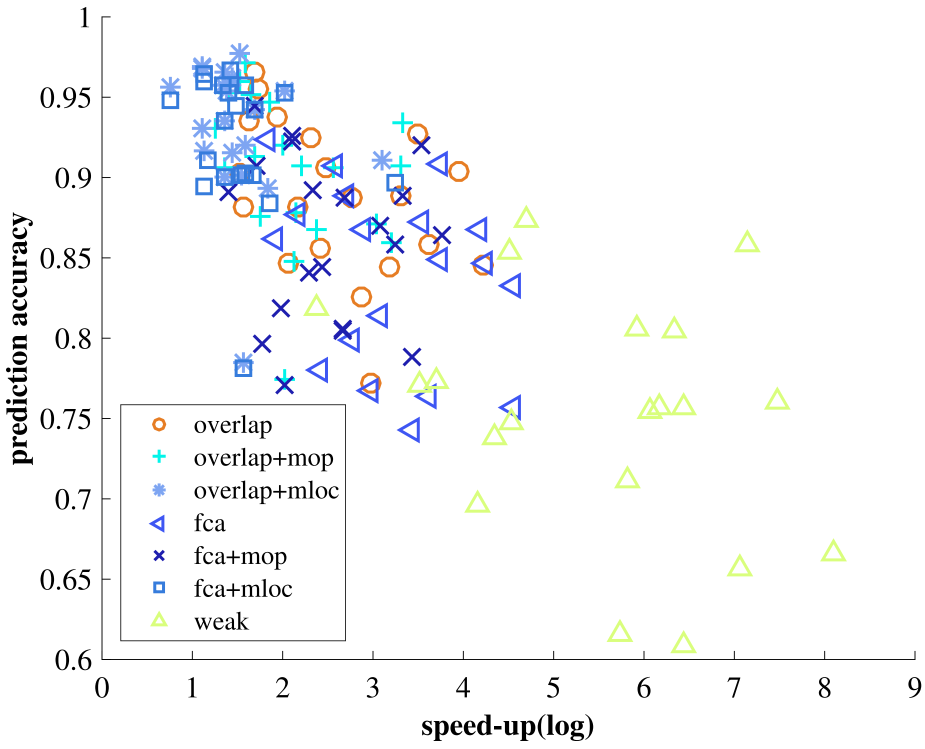
Another important finding is that *mutation location trumps mutation operator in-*

formation when selecting mutants to evaluate for strong mutation. Hence, researchers should take into account the mutation location in addition to the mutation operators when detecting redundant or subsuming mutants (e.g. [46, 202, 219]). This is a clear invitation for future work.

Since our results are encouraging, we envision the following future work: (i) combining mutant location and mutation operator information; (ii) investigating other compression methods, such as Principal Component Analysis [366]; (iii) applying compression techniques in mutation-based test case generation [148, 285].



(a) Speed-up vs. absolute error



(b) Speed-up vs. accuracy

Figure 3.3: Graphical comparison of eight mutation strategies in terms of speed-up, absolute error and accuracy

4

How to Kill Them All: An Exploratory Study on the Impact of Code Observability on Mutation Testing

Mutation testing is well-known for its efficacy to assess test quality, and it is starting to be applied in the industry as well. However, what should a developer do when confronted with a low mutation score? Should the test suite be plainly reinforced (i.e., writing additional tests) to increase the mutation score, or should the production code be improved as well, to make the creation of better tests possible (i.e., improving the production code and then writing additional tests)? In this chapter, we aim to provide a new perspective to developers that enables them to understand and reason about the mutation score in the light of testability *and* observability. *First, we investigate whether testability and observability metrics are correlated with the mutation score on six open-source Java projects. We observe a correlation between observability metrics and the mutation score, e.g.,* test directness, *which measures the extent to which the production code is tested directly, seems to be an essential factor. Based on our insights from the correlation study, we propose a number of "mutation score anti-patterns", which enable software engineers to refactor their existing code or adding tests to be able to improve the mutation score. In doing so, we observe that relatively simple refactoring operations enable an improvement or increase in the mutation score. Moreover, taking Object-Oriented (OO) design principles into consideration, our study can guide developers in making choices between (1) adding new tests, (2) refactoring the production code, or (3) ignoring the surviving mutants when confronting the low mutation score.*

This chapter has been submitted to the Journal of Systems and Software.

4.1. Introduction

Mutation testing has been a very active research field since the 1970s as a technique to evaluate test suite quality in terms of the fault-revealing capability [197]. Recent advances have made it possible for mutation testing to be used in industry [298]. For example, *PIT/PiTest* [99] has been adopted by several companies, such as *The Ladders* and *British Sky Broadcasting* [102]. Furthermore, *Google* [297] has integrated mutation testing with the code review process for around 6000 software engineers.

As mutation testing gains traction in the industry, a better understanding of the *mutation score* (one outcome of mutation testing) becomes essential. The existing works have mainly linked the mutation score with test quality [189, 228] (i.e., *how good is the test suite at detecting faults in the software?*) and *mutant utility* [203, 373] (i.e., *how useful is the mutant?*). However, in Chapter 5, we have observed that certain mutants could be killed only after refactoring the production code to increase the *observability* of state changes. In such cases, test deficiency is not the only reason for the survival mutants, but some issues in the production code, such as *code observability*, result in difficulties to kill the mutants. Different from the previous works (e.g., [189, 203, 228, 373]), our goal is to bring a new perspective to developers that enable them to understand and reason about the mutation score in the light of *testability* and *observability*. Thereby, developers can make a choice when confronting low mutation scores: (1) adding new tests, (2) refactoring the production code to be able to write better tests, or (3) ignoring the surviving mutants.

To this aim, our study consists of two parts: firstly, we investigate the relationship between *testability*/ *observability* and mutation testing in order to find the most correlated metrics; secondly, based on what we observe from the correlations, we define anti-patterns or indicators that software engineers can apply to their code to kill the surviving mutants. More specifically, we start with the investigations of the relationship between *testability/observability metrics* and the mutation score inspired by the work of Bruntink and van Deursen [89]. *Testability* is defined as the "attributes of software that bear on the effort needed to validate the software product" [89, 191]. Given our context, an important part of testability is *observability*, which is a measure of how well internal states of a system can be inferred, usually through the values of its external outputs [328]. Whalen et al. [362] formally defined *observability* as follows: An expression in a program is *observable* in a *test case* if the value of an expression is changed, leaving the rest of the program intact, and the output of the system is changed correspondingly. If there is no such value, the expression is not *observable* for that test. Compared to testability which covers various aspects of a project (e.g., inheritance and cohesion), observability is more specifically addressing the extent to which the value change of expression is observable in a test case.

Our first three research questions steer our investigation in the first part of our study:

RQ4.1 What is the relation between *testability* metrics and the mutation score?

RQ4.2 What is the relation between *observability* metrics and the mutation score?

RQ4.3 What is the relation between the combination of *testability* and *observability* metrics and the mutation score?

After investigating the relationship between *testability*/*observability* and mutation testing, we still lack insight into how these relationships can help developers to take actions when facing survival mutants. That is why, based on the observations from **RQ4.1-RQ4.3**, we define anti-patterns or indicators that software engineers can apply to their code/tests to ensure that mutants can be killed. This leads us to the next research question:

RQ4.4 To what extent does the removal of anti-patterns based on *testability* and *observability* help in improving the mutation score?

In terms of the methodology that we follow in our study, for **RQ4.1-RQ4.3**, we use statistical analysis on open-source Java projects to investigate the relationship between testability, observability, and the mutation score. For **RQ4.4**, we perform a case study with 16 code fragments to investigate whether the removal of anti-patterns increases the mutation score.

4.2. BACKGROUND

In this section, we briefly introduce the basic concepts of and related works on mutation testing, testability metrics, and our proposed metrics for quantifying code observability.

4.2.1. MUTATION TESTING

Mutation testing is defined by Jia and Harman [197] as a fault-based testing technique which provides a testing criterion called the *mutation adequacy score*. This score can be used to measure the effectiveness of a test suite regarding its ability to detect faults [197]. The principle of mutation testing is to introduce syntactic changes into the original program to generate faulty versions (called *mutants*) according to well-defined rules (mutation operators) [279]. The benefits of mutation testing have been extensively investigated and can be summarised in Chapter 2 as 1) having better fault exposing capability compared to other test coverage criteria [144, 228, 244], 2) being a valid substitute to real faults and providing a good indication of the fault detection ability of a test suite [50, 201].

Researchers have actively investigated mutation testing for decades (as evidenced by the extensive survey [197, 241, 279, 393]). Recently, it has started to attract attention from industry [298]. In part, this is due to the growing awareness of the importance of testing in software development [48]. *Code coverage*, the most common metric to measure test suite effectiveness, has seen its limitations being reported in numerous studies (e.g. [144, 189, 228, 244]). Using structural coverage metrics alone might be misleading because in many cases, statements might be covered, but their consequences might not be asserted [189]. Another factor is that a number of well-developed open-source mutation testing tools (e.g., PIT/PiTest [99] and Mull [12]) have contributed to mutation testing being applied in the industrial environments [102, 297, 298].

However, questions still exist about mutation testing, especially regarding the usefulness of a mutant [203]. The majority of the mutants generated by existing mutation operators are equivalent, trivial and redundant [88, 198, 203, 219, 295], which reduces the efficacy of the mutation score. If a class has a high mutation score while most mutants generated are trivial and redundant, the high mutation score does not promise high test effectiveness. A better understanding of mutation score and mutants is thus important.

To address this knowledge gap, numerous studies have investigated how useful mutants are. Example studies include *mutant subsumption* [219], *stubborn* mutants [373], and *real-fault coupling* [201, 295]. These studies paid attention to the context and types of mutants as well as the impact of the test suite, while the impact of production code quality has rarely been investigated. We have seen how code quality can influence how hard it is to test [89] (called software testability [151]), and since mutation testing can generally be considered as "testing the tests," production code quality could also impact mutation testing, just like production code quality has been shown to be correlated with the presence of test smells [325]. Due to the lack of insights into how code quality affects the efforts needed for mutation testing, especially in how to engineer tests that kill *all* the mutants, we conduct this exploratory study. Our study can help researchers and practitioners deepen their understanding of the mutation score, which is generally related to test suite quality and mutant usefulness.

4.2.2. Existing Object-Oriented Metrics for Testability

The notion of *software testability* dates back to 1991 when Freedman [151] formally defined *observability* and *controllability* in the software domain. Voas [354] proposed a dynamic technique coined propagation, infection and execution (PIE) analysis for statistically estimating the program's *fault sensitivity*. More recently, researchers have aimed to increase our understanding of *testability* by using statistical methods to predict *testability* based on various code metrics. Influential works include that of Bruntink and van Deursen [89], in which they explored the relationship between nine object-oriented metrics and testability. To explore the relation between *testability* and mutation score (**RQ4.1**), we first need to collect a number of existing object-oriented metrics which have been proposed in the literature. In total, we collect 64 code quality metrics, including both class-level and method-level metrics that have been the most widely used. We select those 64 metrics because they measure various aspects of a project, including basic characteristics (e.g., NLOC and NOMT), inheritance (e.g., DIT), coupling (e.g., CBO and FIN), and cohesion (LCOM). A large number of those metrics, such as LCOM and HLTH have been widely used to explore software testability [89, 153] and fault prediction [57, 167].

We present a brief summary of the 64 metrics in Table 4.1 (method-level) and Tables 4.2–4.3 (class-level). These metrics have been computed using a static code analysis tool provided by `JHawk` [18].

4.2.3. Code Observability

To explore the relation between *observability* and mutation score (**RQ4.2**), we first need a set of metrics to quantify *code observability*. According to Whalen et al. [362]'s definition of observability (as mentioned in Section 4.1), we consider that *code observability* comprises two perspectives: that of production code and that of the test case. To better explain these two perspectives, let us consider the example in Listing 4.1 from project `jfreechart-1.5.0` showing the method `setSectionPaint` and its corresponding test. This method is to set the section paint associated with the specified key for the `PiePlot` object, and sends a `PlotChangeEvent` to all registered listeners. There is one mutant in Line 3 that removes the call to `org/jfree/chart/plot/PiePlot::fireChangeEvent`.

Table 4.1: Summary of method-level code quality metrics

Abbreviation	Full name	Description
COMP	Cyclomatic Complexity	McCabes cyclomatic Complexity for the method
NOA	Number of Arguments	The number of Arguments
NOCL	Number of Comments	The number of Comments associated with the method
NOC	Number of Comment Lines	The number of Comment Lines associated with the method
VDEC	Variable Declarations	The number of variables declared in the method
VREF	Variable References	The number of variables referenced in the method
NOS	Number of Java statements	The number of statements in the method
NEXP	Number of expressions	The number of expressions in the method
MDN	Max depth of nesting	The maximum depth of nesting in the method
HLTH	Halstead length	The Halstead length of the metric (one of the Halstead Metrics)
HVOC	Halstead vocabulary	The Halstead vocabulary of the method (one of the Halstead Metrics)
HVOL	Halstead volume	The Halstead volume of the method (one of the Halstead Metrics)
HDIF	Halstead difficulty	The Halstead difficulty of the method (one of the Halstead Metrics)
HEFF	Halstead effort	The Halstead effort of the method (one of the Halstead Metrics)
HBUG	Halstead bugs	The Halstead prediction of the number of bugs in the method (one of the Halstead Metrics)
TDN	Total depth of nesting	The total depth of nesting in the method
CAST	Number of casts	The number of class casts in the method
LOOP	Number of loops	The number of loops (for, while) in the method
NOPR	Number of operators	The total number of operators in the method
NAND	Number of operands	The total number of operands in the method
CREF	Number of classes referenced	The classes referenced in the method
XMET	Number of external methods	The external methods called by the method
LMET	Number of local methods	The number of methods local to this class called by this method
EXCR	Number of exceptions referenced	The number of exceptions referenced by the method
EXCT	Number of exceptions thrown	The number of exceptions thrown by the method
MOD	Number of modifiers	The number of modifiers (public, protected, etc.) in method declaration
NLOC	Lines of Code	The number of lines of code in the method

This mutant is not killed by `testEquals`. Looking at the *observability* of this mutant from the *production code* perspective, we can see that the `setSectionPaint` method is void; thus, this mutant is hard to detect because there is no return value for the test case to *assert*. From the *test case* perspective, although `testEquals` invokes the method `setSectionPaint` in Line 14 and 17, no proper assertion statements are used to examine the changes of `fireChangeEvent()` (which is used to send an event to listeners).

Starting with two angles of code observability, we come up with a set of the code observability metrics. Since our study is a starting point to design metrics to measure the code observability, we start with the simple and practical metrics, which are easy for practitioners to understand and apply.

First of all, we consider the return type of the method. As discussed in Listing 4.1, in a void method it is hard to observe the changing states inside the method because there is no return value for test cases to assert. Accordingly, we design two metrics, `is_void` and `non_void_percent` (shown in 1st and 2nd rows in Table 4.5). The metric `is_void` is to examine whether the return value of the method is void or not. The metric `non_void_percent` addresses the return type at class level which measures the percent of non-void methods in the class. Besides these two, a void method might change the

Table 4.2: Summary of class-level code quality metrics (1)

Abbreviation	Full name	Description
NOMT	Number of methods	The number of methods in the class (WMC - one of the Chidamber and Kemerer metrics)
LCOM	Lack of Cohesion of Methods	The value of the Lack of Cohesion of Methods metric for the class. This uses the LCOM* (or LCOM5) calculation. (one of the Chidamber and Kemerer metrics)
TCC	Total Cyclomatic Complexity	The total McCabes cyclomatic Complexity for the class
AVCC	Average Cyclomatic Complexity	The average McCabes cyclomatic Complexity for all of the methods in the class
MAXCC	Maximum Cyclomatic Complexity	The maximum McCabes cyclomatic Complexity for all of the methods in the class
NOS	Number of Java statements	The number of statements in the class
HLTH	Cumulative Halstead length	The Halstead length of the code in the class plus the total of all the Halstead lengths of all the methods in the class
HVOL	Cumulative Halstead volume	The Halstead volume of the code in the class plus the total of all the Halstead volumes of all the methods in the class
HEFF	Cumulative Halstead effort	The Halstead effort of the code in the class plus the total of all the Halstead efforts of all the methods in the class
HBUG	Cumulative Halstead bugs	The Halstead prediction of the number of bugs in the code of the class and all of its methods
UWCS	Un Weighted class Size	The Unweighted Class Size of the class
NQU	Number of Queries	The number of methods in the class that are queries (i.e., that return a value)
NCO	Number of Commands	The number of methods in the class that are commands (i.e., that do not return a value)
EXT	External method calls	The number of external methods called by the class and by methods in the class
LMC	Local method calls	The number of methods called by the class and by methods in the class
HIER	Hierarchy method calls	The number of local methods called by the class and by methods in the class that are defined in the hierarchy of the class
INST	Instance Variables	The number of instance variables declared in the class
MOD	Number of Modifiers	The number of modifiers (public, protected, etc.) applied to the declaration of the class
INTR	Number of Interfaces	The number of interfaces implemented by the class

field(s) of the class it belongs to. A workaround to test a *void* method is to invoke getters. So `getter_percentage` (shown in 3rd row in Table 4.5) is proposed to complement `is_void`.

Secondly, we come up with the access control modifiers. Let us consider the example in Listing 4.2 from project `commons-lang-LANG_3_7`. The method `getMantissa` in class `NumberUtils` returns the mantissa of the given number. This method has only one mutant: the return value is replaced with "`return if (getMantissa(str, str.length()) != null) null else throw new RuntimeException`"[1]. This mutant should be easy to detect given an input of either a legal String object (the return value is not null) or a null string (throw an exception). The reason this "trivial" mutant is not detected is because the method `getMantissa` is private. The access control modifier *private* makes it impossible to *directly* test the method `getMantissa`, for this method is only visible to methods from class `NumberUtils`. To test this method, the test case must first invoke a

[1]This mutant is generated by `Return Values Mutator` in PIT [103]. In Listing 4.2, getMantissa(str, str.length()) returns a `String` object. When the return value of a method is an object, the mutator replaces non-null return values with null and throw a `java.lang.RuntimeException` if the un-mutated method would return null.

Table 4.3: Summary of class-level code quality metrics (2)

Abbreviation	Full name	Description
PACK	Number of Packages imported	The number of packages imported by the class
RFC	Response for Class	The value of the Response For Class metric for the class. (One of the Chidamber and Kemerer metrics)
MPC	Message passing	The value of the Message passing metric for the class
CBO	Coupling between objects	The value of the Coupling Between Objects metric for the class. (One of the Chidamber and Kemerer metrics)
FIN	Fan In	The value of the Fan In (Afferent coupling (Ca)) metric for the class
FOUT	Fan Out	The value of the Fan Out (Efferent coupling (Ce)) metric for the class
R-R	Reuse Ratio	The value of the Reuse Ratio for the class
S-R	Specialization Ratio	The value of the Specialization Ratio for the class
NSUP	Number of Superclasses	The number of superclasses (excluding Object) in the hierarchy of the class
NSUB	Number of Subclasses	The number of subclasses below the class in the hierarchy. (NOC - one of the Chidamber and Kemerer metrics)
MI	Maintainability Index (including comments)	The Maintainability Index for the class, including the adjustment for comments
MINC	Maintainability Index (not including comments)	The Maintainability Index for the class without any adjustment for comments
COH	Cohesion	The value of the Cohesion metric for the class
DIT	Depth of Inheritance Tree	The value of the Depth of Inheritance Tree metric for the class. (One of the Chidamber and Kemerer metrics)
LCOM2	Lack of Cohesion of Methods (variant 2)	The value of the Lack of Cohesion of Methods (2) metric for the class.This uses the LCOM2 calculation. (One of the Chidamber and Kemerer metrics)
CCOM	Number of Comments	The number of Comments associated with the class
CCML	Number of Comment Lines	The number of Comment Lines associated with the class
cNLOC	Lines of Code	The number of lines of code in the class and its methods

method that calls method `getMantissa`. From this case, we observe that access control modifiers influence the *visibility* of the method, so as to play a significant role in code observability. Thereby, we take access control modifiers into account to quantify code observability, where we design `is_public` and `is_static` (shown in 4th and 5th rows in Table 4.5).

The third point we raise concerns fault masking. We have observed that mutants generated in *certain locations* are more likely to be *masked* [162], i.e., the state change cannot propagate to the output of the method. The first observation is that mutants that reside in a nested class. The reasoning is similar to mutants that reside in nested sections of code, namely that a change in intermediate results does not propagate to a point where a test can pick it up. Thus, we come up with `is_nested` (in 6th row in Table 4.5). Another group of mutants is generated inside nested conditions and loops. These can be problematic because the results of the mutations cannot propagate to the output, and the tests have no way of checking the intermediate results within the method. Accordingly, we define `nested_depth` (shown in 7th row in Table 4.5) and a set of metrics to quantify the conditions and loops (shown in 8th through 13th rows in Table 4.5). The last observation is related to mutants that are inside a long method (the reason is similar to the mutants inside nested conditions and loops), thus we design `method_length` (shown in 14th row in Table 4.5).

The next aspect we consider is test directness. Before we dig into test directness, we

```java
public void setSectionPaint(Comparable key, Paint paint) {
  this.sectionPaintMap.put(key, paint);
  fireChangeEvent(); // mutant: remove this method
}

@Test
public void testEquals() {
...
  PiePlot plot1 = new PiePlot();
  PiePlot plot2 = new PiePlot();
  assertTrue(plot1.equals(plot2));
  assertTrue(plot2.equals(plot1));
  // sectionPaintMap
  plot1.setSectionPaint("A", new GradientPaint(1.0f, 2.0f,
    Color.BLUE,3.0f, 4.0f, Color.WHITE));
  assertFalse(plot1.equals(plot2));
  plot2.setSectionPaint("A", new GradientPaint(1.0f, 2.0f,
    Color.BLUE,3.0f, 4.0f, Color.WHITE));
  assertTrue(plot1.equals(plot2));
...
}
```

Listing 4.1: Example of Method `setSectionPaint` in Class `PiePlot` and its test

```java
  private static String getMantissa(final String str) {
    return getMantissa(str, str.length());
  }
```

Listing 4.2: Example of Method `getMantissa` in Class `NumberUtils`

take Listing 4.3 as an instance. Listing 4.3 shows the class `Triple` from project `commons-lang-LANG_3_7`, which is an abstract implementation defining the basic functions of the object and that consists of three elements. It refers to the elements as “left”, “middle” and “right”. The method `hashCode` returns the hash code of the object. Six mutants are generated for the method `hashCode` in class `Triple`. Table 4.4 summarises all the mutants from Listing 4.3. Of those six mutants, only Mutant 1 is killed, and the other mutants are not equivalent. Through further investigation of method `hashCode` and its test class, we found that although this method has 100% coverage by the test suite, there is no *direct* test for this method. A *direct* test would mean that the test method directly invoking the method (production code) [59]. The direct test is useful because it allows to control the input data directly and to assert the output of a method directly. This example shows that test directness can influence the outcome of mutation testing, which denotes the test case angle of *code observability*. Previous works such as Huo and Clause [187] also addressed the significance of test directness in mutation testing. Therefore, we design two metrics, `direct_test_no.` and `test_distance` (shown in 15th and 16th row in Table 4.5), to quantify test directness. Those two metrics represent the *test case* perspective of code observability.

Last but not least, we take assertions into considerations. As discussed in Listing 4.1, we have observed that mutants without appropriate assertions in place (throwing exceptions is also under consideration) cannot be killed, as a prerequisite to killing a mutant is to have the tests fail in the mutated program. Schuler and Zeller [315] and Zhang and

```
@Override
public int hashCode() {
  return (getLeft() == null ? 0 : getLeft().hashCode()) ^
    (getMiddle() == null ? 0 : getMiddle().hashCode()) ^
    (getRight() == null ? 0 : getRight().hashCode());
}
```

Listing 4.3: Example of Method `hashCode` in Class `Triple`

Table 4.4: Summary of mutants from Listing 4.3

ID	Line No.	Mutator	Results
1	3	negated conditional	Killed
2	3	replaced return of integer sized value with (x == 0 ? 1 : 0)	Survived
3	3	Replaced XOR with AND	Survived
4	4	negated conditional	Survived
5	4	Replaced XOR with AND	Survived
6	5	negated conditional	Survived

Mesbah [384] also drew similar conclusion to ours. Accordingly, we come up with three metrics to quantify assertions in the method, `assertion_no.`, `assertion-McCabe_Ratio` and `assertion_density` (shown in 17th - 19th rows in Table 4.5). The `assertion-McCabe_Ratio` metric [59] is originally proposed to measure *test completeness* by indicating the ratio between the number of the actual points of testing in the test code and the number of decision points in the production code (i.e., how many decision points are tested). For example, a method has a McCabe complexity of 4, then in the ideal case, we would expect 4 different assertions to test those linear independent paths (in this case this ration would be 1), but if the ratio is lower than 1, it could be an indication that either not all paths are tested, or that not all paths are tested in a direct way. The `assertion_density` metric [217] aims at measuring the ability of the test code to detect defects in the parts of the production code that it covers. We include those two metrics here as a way to measure the quality of assertions. These three metrics are proposed based on the *test case* perspective of code observability.

To sum up, Table 4.5 presents all the code observability metrics we propose, where we display the name, the definition of each metric, and the category.

4.3. EXPERIMENTAL SETUP

To examine our conjectures, we conduct an experiment using six open-source projects. We recall the research questions we have proposed in Section 4.1:

- **RQ4.1**: *What is the relation between* testability *metrics and the mutation score?*
- **RQ4.2:** *What is the relation between* observability *metrics and the mutation score?*
- **RQ4.3:** *What is the relation between the combination of* testability *and* observability *metrics and the mutation score?*
- **RQ4.4:** *To what extent does removal of anti-patterns based on testability and observability help in improving the mutation score?*

Table 4.5: Summary of code observability metrics

#	Name	Definition	Category
1	is_void	whether the return value of the method is void or not	return type
2	non_void_percent (class-level)	the percent of non-void methods in the class	
3	getter_percentage	the percentage of getter methods in the class[1]	
4	is_public	whether the method is public or not	access control modifiers
5	is_static	whether the method is static or not	
6	is_nested (class-level)	whether the method is located in a nested class or not	fault masking
7	nested_depth	the maximum number of nested depth (`MDN` from Section 4.2.2)	
8	(cond)	the number of conditions (`if`, `if-else` and `switch`) in the method	
9	(cond(cond))	the number of nested conditions (e.g.,`if{if{}}`) in the method	
10	(cond(loop))	the number of nested condition-loops (e.g.,`if{for{}}`) in the method	
11	(loop)	the number of loops (`for`, `while` and `do-while`) in the method (`LOOP` from Section 4.2.2)	
12	(loop(cond))	the number of nested loop-conditions (e.g.,`for{if{}}`) in the method.	
13	(loop(loop))	the number of nested loop-conditions (e.g.,`for{for{}}`) in the method.	
14	method_length	the number of lines of code in the method (`NLOC` from Section 4.2.2)	
15	direct_test_no.	the number of test methods directly invoking the method under test (production code)[2]	test directness
16	test_distance	the shortest method call sequence required to invoke the method (production code) by test methods[3]	
17	assertion_no.	the number of assertions in direct tests	assertion
18	assertion-McCabe_Ratio	the ratio between the total number of assertions in direct tests and the McCabe Cyclomatic complexity	
19	assertion_density	the ratio between the total number of assertions in direct tests and the lines of code in direct tests	

[1]A getter method must follow three patterns [49]: (1) must be public; (2) has no arguments and its return type must be something other than void. (3) have naming conventions: the name of a getter method begins with "get" followed by an uppercase letter.
[2]If the method is not directly tested, then its direct_test_no. is 0.
[3]If the method is directly tested, then its test_distance is 0. The maximum test_distance is set `Integer.MAX_VALUE` in Java which means there is no method call sequence that can reach the method from test methods.

4.3.1. Mutation Testing

We adopt PIT (Version 1.4.0) [99] to apply mutation testing in our experiments. The mutation operators we adopt are the *default* mutation operators provided by PIT [103]: `Conditionals Boundary Mutator`, `Increments Mutator`, `Invert Negatives Mutator`, `Math Mutator`, `Negate Conditionals Mutator`, `Return Values Mutator` and `Void Method Calls Mutator`. We did not adopt the extended set of mutation operators provided PIT, as the operators in the default version are largely designed to be stable (i.e., not be too easy to detect) and minimise the number of equivalent mutations that they

Table 4.6: Subject systems

PID	Project	LOC	#Tests	#Methods		#Mutants	
				#Total	#Selected	#Total	#Killed
1	Bukkit-1.7.9-R0.2	32373	432	7325	2385	7325	947
2	commons-lang-LANG_3_7	77224	4068	13052	2740	13052	11284
3	commons-math-MATH_3_6_1	208959	6523	48524	6663	48524	38016
4	java-apns-apns-0.2.3	3418	91	429	150	429	247
5	jfreechart-1.5.0	134117	2175	34488	7133	34488	11527
6	pysonar2-2.1	10926	269	3070	719	3074	836
	Overall	467017	13558	106888	19790	106892	62857

Note: Column "LOC" standing for the line of code is measured by sloccount[363]. In our experiment, we remove the methods with no generated mutant by PIT, thus resulting in the number of selected methods (#Selected).

generate [103].

4.3.2. SUBJECT SYSTEMS

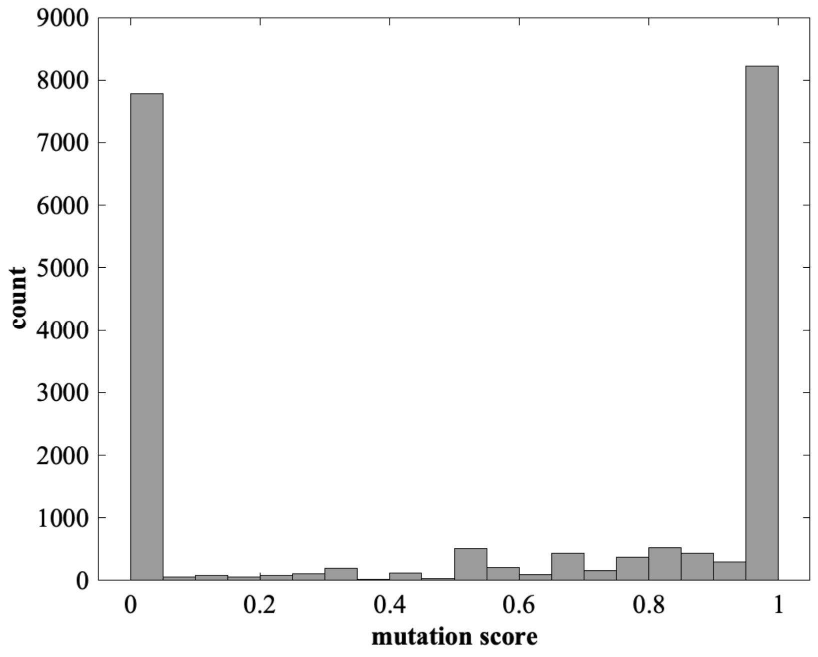
We use six systems publicly available on GitHub in this experiment. Table 4.6 summarises the main characteristics of the selected projects, which include the lines of code (LOC), the number of tests (#Test), the total number of methods (#Total Methods), the number of selected methods used in our experiment (#Selected), the total number of mutants (#Total Mutants), and the killed mutants (#Killed). In our experiment, we remove the methods with no generated mutant by PIT, thus resulting in the number of selected methods (#Selected). These systems are selected because they have been widely used in the research domain (e.g., [187, 315, 378, 384, 392]). All systems are written in Java, and tested by means of JUnit. The granularity of our analysis is at method-level.

The results of the mutants that are killable for all of the subjects are shown in Columns 7-8 of Table 4.6. Figure 4.1a shows the distribution of mutation score among selected methods. The majority of the mutation scores are either 0 or 1. Together with Figure 4.1b, we can see that the massive number of 0s and 1s are due to the low mutant number per method. Most methods show less than 10 mutants, which is mainly due to most methods being short methods (`NOS` < 2 as shown in Figure 4.2). Writing short methods is a preferred strategy in practice, for a long method is a well-known code smells [70]. Besides, PIT adopts several optimisation mechanisms [101] to reduce the number of mutants. Thus, the number of mutants (#Total Mutants) shown in Table 4.6 is fewer than the actual number of generated mutants. The large number of the methods with low mutant number is an unavoidable bias in our experiment.

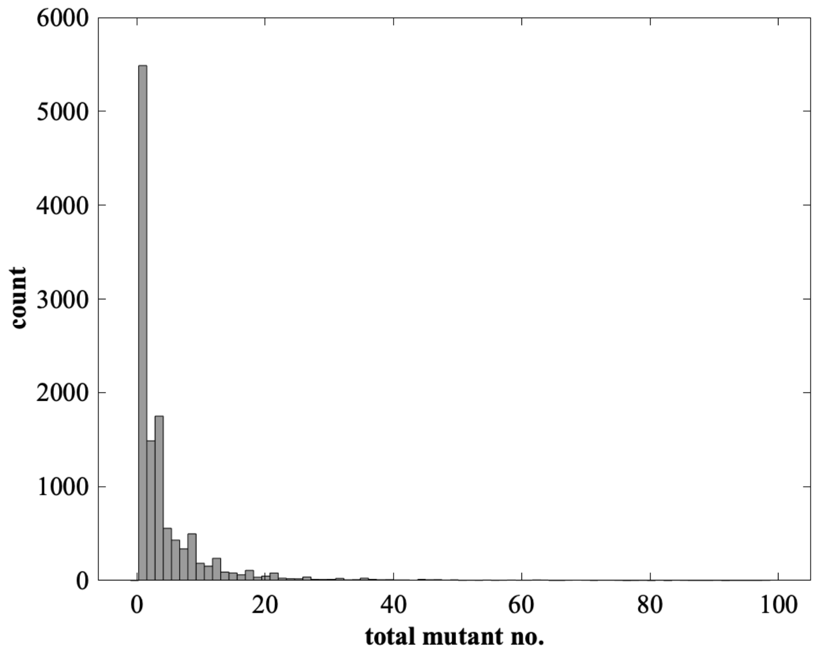
4.3.3. TOOL IMPLEMENTATION

To evaluate the *code observability metrics* that we have proposed, we implement a prototype tool (coined MUTATION OBSERVER) to capture all the necessary information from both the program under test and the mutation testing process. This tool is openly available on GitHub [390].

Our tool extracts information from three parts of the system under test (in Java): source code, bytecode, and tests. Firstly, Antlr [1] parses the source code to obtain the basic code features, e.g., **is public**, **is static** and **(cond)**. Secondly, we adopt Apache Com-



(a) Distribution of mutation score per method



(b) Distribution of total mutant No. per method

Figure 4.1: Distribution of mutation score and mutant No.

mons BCEL [2] to parse the bytecode. Then, `java-callgraph` [17] generates the pairs of method calls between the source code and tests, which we later use to calculate **direct**

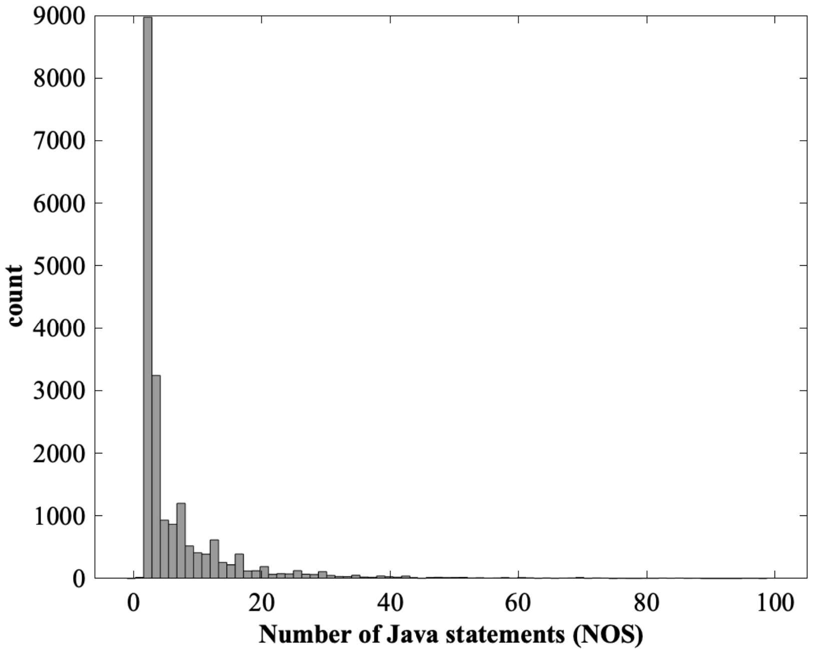


Figure 4.2: Distribution of Number of Java Statements (NOS) per method

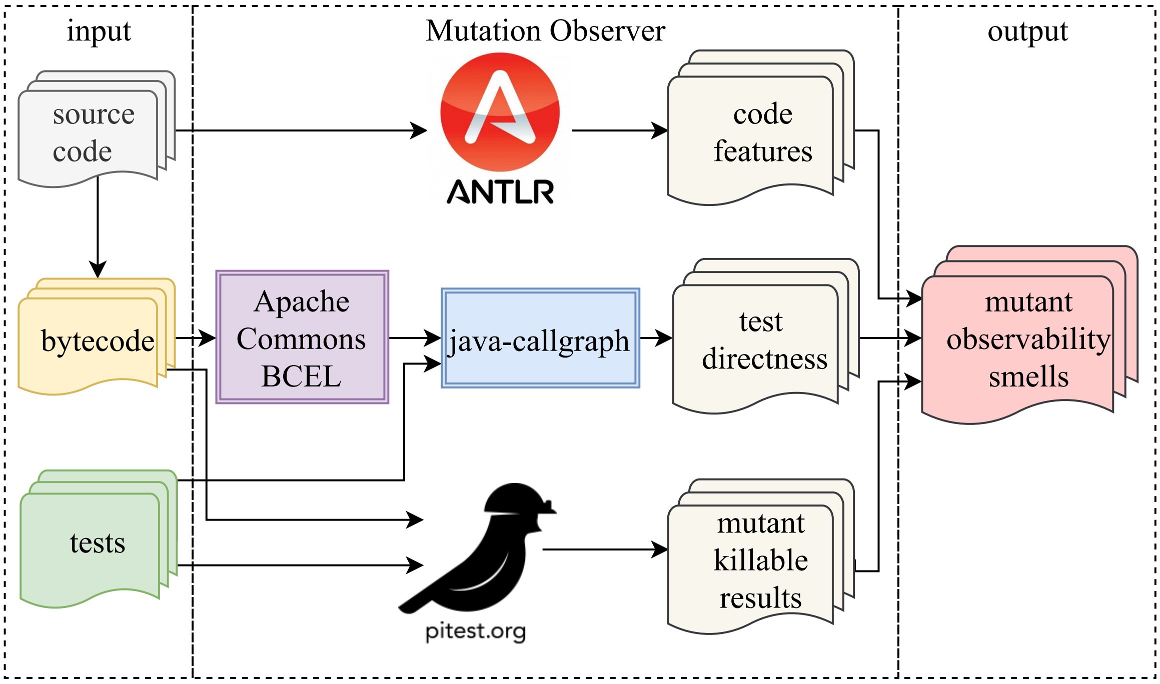


Figure 4.3: Overview of Mutation Observer architecture

test no. and other test call related metrics. The last part is related to the mutation testing process, for which we adopt PIT (Version 1.4.0) [99] to obtain the killable mutant results. An overview of the architecture of Mutation Observer can be seen in Figure 6.4.

4.3.4. Design of Experiment

RQ4.1-RQ4.3

Our investigation of the relationships between testability/observability metrics and the mutation score (**RQ4.1-RQ4.3**) is two-fold: in the first part, we adopt Spearman's rank-order correlation to measure the *pairwise correlations* statistically between each metric (both testability and observability metrics) and the mutation score; in the second part, we turn the correlation problem into a binary classification problem (where we adopt Random Forest as the classification algorithm) to investigate how those metrics *interact* with one another.

Pairwise correlations To answer **RQ4.1**, **RQ4.2**, and **RQ4.3**, we first adopt Spearman's rank-order correlation to statistically measure the correlation between each metric (both testability and observability metrics) and the mutation score of the corresponding methods or classes. Spearman's correlation test checks whether there exists a monotonic relationship (linear or not) between two data samples. It is a *non-parametric* test and, therefore, it does not make any assumption about the distribution of the data being tested. The resulting coefficient ρ takes values in the interval $[-1;+1]$; the higher the correlation in either direction (positive or negative), the stronger the monotonic relationship between the two data samples under analysis. The strength of the correlation can be established by classifying into "negligible" ($|\rho| < 0.1$), "small" ($0.1 \leq |\rho| < 0.3$), "medium" ($0.3 \leq |\rho| < 0.5$), and "large" ($|\rho| \geq 0.5$) [178]. Positive ρ values indicate that one distribution increases when the other increases as well; negative ρ values indicate that one distribution decreases when the other increases.

The mutation score[2] is calculated by Equation 4.1 (method-level).

$$mutation\,score\,(A) = \frac{\#\,killed\,mutants\,in\,method\,A}{\#\,total\,mutants\,in\,method\,A} \tag{4.1}$$

We adopt `Matlab` [246] to calculate the Spearman's rank-order correlation coefficient between each metric and the mutation score. In particular, we used the statistical analysis (`corr` function with the option of "`Spearman`" in Matlab's default package[3]).

Interactions Except for the pairwise correlations between each metric and the mutation score, we are also interested in how those metrics interact with one another. To do so, we first turn the correlation problem into a binary classification problem. We use 0.5 as the cutoff between `HIGH` and `LOW` mutation core because 0.5 is widely used as a cutoff in classification problems whose independent variable ranges in [0,1] (e.g., defect prediction [340, 377]). We consider all the metrics to predicate whether the method belongs to classes with `HIGH` or `LOW` mutation score. One thing to notice here is that building a perfect prediction model is not our primary goal. Our interest is to see which metrics and/or which combinations of the metrics contributing to the LOW mutation score by

[2] In the original equation for mutation score, the divisor is the number of non-equivalent. In our study, our main focus is the relation between testability/observability metrics and mutation score, rather than mutation score itself. In our previous literature review (Chapter 2), we have found that treating all mutants as non-equivalent is a common method when the mutation score is used as a relative comparison. Therefore, we do not manually analyse the equivalent mutants, and treated all mutants as non-equivalent.

[3] `https://www.mathworks.com/help/stats/corr.html`

building the prediction models. Therefore, deciding different thresholds values is outside the scope of this chapter.

For prediction, we adopt *Random Forest* [83] as the classification algorithm, where we use `WEKA` [143] to build the prediction model. Random Forest is an ensemble method based on a collection of decision tree classifiers, where the individual decision trees are generated using a random selection of attributes at each node to determine the split [169]. Besides, Random Forest is more accurate than one decision tree, and it is not affected by the overfitting problem [169]. The reasons why we adopted Random Forest rather than linear regression here are as follows: (1) Random Forest is a more advanced method, and it can usually achieve high accuracy and avoid overfitting when considering many attributes. (2) We did build the linear regression in the first place. As the linear regression model enables to predict a certain value (i.e., the mutation score), it incurs far higher cross-validation errors than Random Forest.

As our investigation includes testability and observability metrics, for each project, we compare three types of classification models: (1) a model based on merely existing testability metrics, (2) a model based on merely code observability metrics, and (3) a model based on the combination of existing and our observability metrics (overlapping metrics, e.g., `method_length` to `NLOC`, are only considered once). In particular, we include the model based on the combination of the two aspects for further comparison: to see whether the combination of the two aspects can work better than each aspect itself. To examine the effectiveness of Random Forest in our dataset, we also consider `ZeroR`, which classifies all the instances to the majority and ignores all predictors, as the baseline. It might be that our data is not balanced, as in that one project has over 90% methods with a `HIGH` mutation score. This could entail that the classification model achieving 90% accuracy is not necessarily an effective model. In this situation, `ZeroR` could also achieve over 90% accuracy in that scenario. Our Random Forest model must thus perform better than `ZeroR`; otherwise, the Random Forest model is not suitable for our dataset.

In total, we consider four classification models: 1) `ZeroR` (i.e., the constant classifier), 2) Random Forest based on existing metrics, 3) Random Forest based on code observability metrics, and 4) Random Forest based on the combination of existing metrics and code observability metrics. To build Random Forest, `WEKA` [143] adopts bagging in tandem with random attribute selection. We use WEKA's default parameters to train the Random Forest model, i.e., "-P 100 -I 100 -num-slots 1 -K 0 -M 1.0 -V 0.001 -S 1". To evaluate the performance of the classifier model (e.g., precision and recall), we use 10-fold cross validation [214].

In terms of feature importance, we apply `scikit-learn` [296] to conduct the analysis. To determine the feature importance, `scikit-learn` [296] implements "*Gini Importance*" or "*Mean Decrease Impurity*" [84]. The importance of each feature is computed by the probability of reaching that node (which is approximated by the proportion of samples reaching that node) averaged over total tree ensembles [84]. We use the method of `feature_importances_` in `sklearn.ensemble.RandomForestRegressor` [4] package to analyse the feature importance.

[4]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestRegressor.html

RQ4.4

To answer **RQ4.4**, we first need to establish the anti-patterns (or smells) based on these metrics. An example of an anti-pattern rule generated from the metrics is `method_length` > 20 and `test_distance` > 2. In this case, it is highly likely that the method has low mutation score. To obtain the anti-pattern rules, we adopt J48 to build a decision tree [143, 301]. We consider J48 because of its advantage in interpretation over Random Forest. After building the decision tree, we rank all leaves (or paths) according to instances falling into each leaf and accuracy. We select the leaves with the highest instances and accuracy ≥ 0.8 for further manual analysis, to understand to what extent refactoring of the anti-patterns can help in improving the mutation score.

4.3.5. EVALUATION METRICS

For **RQ4.1**, **RQ4.2**, and **RQ4.3**, to ease the comparisons of the four classification models, we consider four metrics which have been widely used in classification problems: precision, recall, AUC, and the mean absolute error. To that end, we first introduce four key notations: TP, FP, FN, and TN, which denotes true positive, false positive, false negative, and true negative, respectively.

In our case, we cannot decide which class is positive or not, or in other words, we cannot say HIGH mutation score is what we expect. The goal to use a prediction model is to investigate the interactions between those metrics or how they interact with each other. So we adopt *weighted* precision and recall, which also take the number of instances in each class into consideration.

Weighted precision. The precision is the fraction of true positive instances in the instances that are predicted to be positive: TP/(TP+FP). The higher the precision, the fewer false positives. The *weighted precision* is computed as follows, where p_{c1} and p_{c2} are the precisions for class 1 and class 2, and $|c1|$ and $|c2|$ are the number of instances in class 1 and class 2, respectively:

$$weighted\,precision = \frac{p_{c1} \times |c1| + p_{c2} \times |c2|}{|c1| + |c2|} \tag{4.2}$$

Weighted recall. The recall is the fraction of true positive instances in the instances that are actual positives: TP/(TP+FN). The higher the recall, the fewer false negative errors there are. The *weighted recall* is computed as follows, where r_{c1} and r_{c2} are the recalls for class 1 and class 2, and $|c1|$ and $|c2|$ are the number of instances in class 1 and class 2:

$$weighted\,recall = \frac{r_{c1} \times |c1| + r_{c2} \times |c2|}{|c1| + |c2|} \tag{4.3}$$

AUC. The area under ROC curve, which measures the overall discrimination ability of a classifier. An area of 1 represents a perfect test; an area of 0.5 represents a worthless test.

Mean absolute error. The mean of overall differences between the predicted values and actual values.

Table 4.7: Spearman results of existing code metrics for testability

metric	**rho**	**p-value**	**metric**	**rho**	**p-value**	**metric**	**rho**	**p-value**
COMP	0.0398	2.16E-08	NOC	0.1908	1.254E-161	R-R(class)	**-0.2524**	3.721E-285
NOCL	0.1047	2.32E-49	NOA	0.0423	2.723E-09	NSUB(class)	-0.0048	0.5009
NOS	-0.0139	0.05024	CAST	-0.0162	0.02302	NSUP(class)	**-0.2634**	0
HLTH	0.0518	2.927E-13	HDIF	0.1334	2.691E-79	NCO(class)	-0.0751	3.602E-26
HVOC	0.0485	8.831E-12	NEXP	0.0288	5.135E-05	FOUT(class)	-0.1073	9.482E-52
HEFF	0.0856	1.595E-33	NOMT(class)	0.0981	1.564E-43	DIT(class)	**-0.2634**	0
HBUG	0.0518	3.163E-13	LCOM(class)	0.0564	2.125E-15	CCOM(class)	0.1695	1.589E-127
CREF	0.0193	0.00653	AVCC(class)	0.0405	1.206E-08	COH(class)	0.0001	0.9852
XMET	0.0465	5.743E-11	NOS(class)	0.0793	5.416E-29	S-R(class)	0.0016	0.8184
LMET	-0.0221	0.00191	HBUG(class)	0.0824	3.826E-31	MINC(class)	-0.0255	0.0003272
NLOC	-0.0004	0.95	HEFF(class)	0.0982	1.213E-43	EXT(class)	-0.0636	3.314E-19
VDEC	0.0281	7.702E-05	UWCS(class)	0.0929	3.708E-39	INTR(class)	-0.0571	9.413E-16
TDN	0.0408	9.634E-09	INST(class)	0.0045	0.5238	MPC(class)	-0.0636	3.314E-19
NAND	0.0357	5.191E-07	PACK(class)	-0.1029	9.956E-48	HVOL(class)	0.0823	4.344E-31
LOOP	0.0685	5.116E-22	RFC(class)	0.095	6.38E-41	HIER(class)	**-0.212**	6.066E-200
MOD	0.0103	0.1482	CBO(class)	-0.0157	0.0274	HLTH(class)	0.0911	9.53E-38
NOPR	0.067	3.801E-21	MI(class)	0.0482	1.144E-11	SIX(class)	-0.197	2.388E-172
EXCT	0.1125	9.723E-57	CCML(class)	0.1559	6.998E-108	TCC(class)	0.0897	1.203E-36
MDN	0.053	8.3E-14	NLOC(class)	0.0756	1.692E-26	NQU(class)	0.1489	1.568E-98
EXCR	-0.0067	0.3473	RVF(class)	-0.033	3.498E-06	F-IN(class)	0.0875	6.031E-35
HVOL	0.0512	5.719E-13	LCOM2(class)	-0.0486	7.691E-12	MOD(class)	0.0516	3.738E-13
VREF	0.0446	3.42E-10	MAXCC(class)	-0.0178	0.01245	LMC(class)	0.1034	3.68E-48

Note: Column“rho” represents the pairwise correlation coefficient between each code metric and the mutation score. Column“p-value" denotes the strength of evidence for testing the hypothesis of no correlation against the alternative hypothesis of a non-zero correlation using Spearman's rank-order. For p-values, we use 0.05 as the cutoff for significance. For convenience, we highlighted the p-values higher than 0.05 by underlining; those correlation results are not statistically significant. Also, we highlight the top 4 metrics in **bold** face in terms of absolute value of rho.

4.4. RQ4.1 - RQ4.3 TESTABILITY VERSUS OBSERVABILITY VERSUS COMBINATION

We opt to discuss the three research questions, **RQ4.1**, **RQ4.2**, and **RQ4.3**, together, because it gives us the opportunity to compare testability, observability, and their combination in detail.

4.4.1. SPEARMAN'S RANK ORDER CORRELATION

TESTABILITY

Findings Table 4.7 presents the overall results of Spearman's rank-order correlation analysis for existing code metrics. The columns of “rho” represent the pairwise correlation coefficient between each code metric and the mutation score. The *p-values* columns denote the strength of evidence for testing the hypothesis of no correlation against the alternative hypothesis of a non-zero correlation using Spearman's rank-order. Here we used 0.05 as the cutoff for significance. From Table 4.7 , we can see that except for `NOS`, `NLOC`, `MOD`, `EXCR`, `INST(class)`, `NSUB(class)`, `COH(class)` and `S-R(class)` (which, for convenience, we highlighted by underlining the value), the correlation results for the metrics are all statistically significant.

Overall, the pairwise correlation between each source code metric and the mutation score is not strong ($|rho| < 0.27$). We speculate the reason behind the weak correlations to be collinearity of these code metrics. More specifically, Spearman's rank-order correlation analysis only evaluates the correlation between individual code metric and mutation score. Some code metrics could interact with one another. For example, a long method does not necessarily have a low mutation score. Alternatively, another example: if there are more than four loops in a long method, then the method is very likely to have low mutation score. That is also an example of the collinearity, i.e., the number of loops and the method length are highly correlated.

From Table 4.7, we can see that the highest *rho*[5] is -0.2634 for both `NSUP (class)` standing for Number of Superclasses, and `DIT(class)`, or Depth of Inheritance Tree. Followed by `R-R(class)`, for Reuse Ratio, and `HIER(class)`, for Hierarchy method calls. At first glance, the top 4 metrics are all class-level metrics. However, we cannot infer that class-level metrics are more impactful on the mutation score than method-level ones. In particular, it can be related to the fact that we have considered more class-level metrics than method-level ones in the experiment.

Additionally, we expected that the metrics related to McCabe's Cyclomatic Complexity, i.e., `COMP`, `TCC`, `AVCC` and `MAXCC` would show stronger correlation to the mutation score. In fact, McCabe's Cyclomatic Complexity has been widely considered as a powerful measure to quantify the complexity of a software program, and it is used to provide a lower bound to the number of tests that should be written [140, 157, 369]). Based on our results without further investigation, we could only speculate that McCabe's Cyclomatic Complexity might not directly influence the mutation score.

Summary *We found that the pair-wise correlations between the 64 existing source code metrics and the mutation score to be not so strong ($|rho| < 0.27$). The top 4 metrics with the strongest correlation coefficients are `NSUP(class)`, `DIT(class)`, `R-R(class)` and `HIER(class)`.*

OBSERVABILITY

Findings Table 4.8 shows the overall results of Spearman's rank-order correlation analysis for code observability metrics. From Table 4.8, we can see that except for `method_length` and `(cond(loop))`, whose *p-value* is greater than 0.05, the results of the other observability metrics are statistically significant. The overall correlation between code observability metrics and mutation score is still not strong (<0.5), but significantly better than existing code metrics (<0.27). The top five metrics are `test_distance`, `direct_test_no.`, `assertion-density`, `assertion-McCabe` and `assertion_no`. The metrics related to *test directness*, i.e., `test_distance` (-0.4923) and `direct_test_no` (0.4177) are ranked first in terms of *rho* among all metrics that we consider (including existing code metrics in Section 4.2.2). This observation corresponds to our hypothesis in Section 4.2.3 that the methods with no direct tests are more challenging to kill mutants. In terms of *rho* values, the assertion related metrics are ranked after test directness related metrics; this confirms both our conjectures in Section 4.2.3 and what has been reported in the related literature [315, 384] that the quality of assertions can influence the outcome of mutation testing.

[5]In terms of absolute value.

Table 4.8: Spearman results of code observability metrics

metric	rho	p-vlaue	metric	rho	p-vlaue
is_public	-0.0639	2.35E-19	(cond(cond))	-0.0415	5.4E-09
is_static	0.1137	6.29E-58	(cond(loop))	0.0073	0.302
is_void	-0.1427	1.42E-90	(loop)	0.0685	5.12E-22
is_nested	0.0466	5.38E-11	(loop(cond))	0.0216	0.00242
method_length	-0.0004	0.95	(loop(loop))	0.0428	1.65E-09
nested_depth	0.053	8.3E-14	non_void_percent	0.2424	1.24E-262
direct_test_no	**0.4177**	0	getter_percent	-0.153	6.23E-104
test_distance	**-0.4921**	0	assertion-McCabe	**0.3956**	0
assertion_no	**0.3858**	0	assertion-density	**0.4096**	0
(cond)	0.023	0.00124			

Note: Column“rho” represents the pairwise correlation coefficient between each code metric and the mutation score. Column“p-value" denotes the strength of evidence for testing the hypothesis of no correlation against the alternative hypothesis of a non-zero correlation using Spearman's rank-order. For p-values, we use 0.05 as the cutoff for significance. For convenience, we highlighted the p-values higher than 0.05 by underlining; those correlation results are not statistically significant. Also, we highlight the top 5 metrics in **bold** face in terms of absolute value of rho.

Table 4.9: Random forest results of code observability metrics vs. existing metrics

pid	ZeroR				existing				code observability				combined			
	prec.	recall	AUC	err.	prec.	recall	AUC	err.	prec.	recall	AUC	err.	prec.	recall	AUC	err.
1	-	0.856	0.497	0.2465	0.927	0.93	0.961	0.1014	0.940	0.942	0.960	0.0786	**0.946**	**0.948**	**0.976**	**0.0741**
2	-	0.913	0.498	0.1595	0.947	0.951	0.932	0.0775	**0.960**	**0.962**	0.946	**0.063**	0.957	0.959	**0.951**	0.067
3	-	0.815	0.499	0.3015	0.848	0.861	0.836	0.2039	0.866	0.864	0.871	0.1727	**0.887**	**0.893**	**0.909**	**0.167**
4	-	0.507	0.468	0.5001	0.667	0.667	0.733	0.3831	**0.861**	**0.860**	**0.909**	**0.2044**	0.827	0.827	0.887	0.2626
5	-	0.62	0.5	0.4712	0.842	0.843	0.908	0.2347	0.868	0.869	0.931	0.1801	**0.901**	**0.901**	**0.955**	**0.168**
6	-	0.726	0.493	0.3982	0.73	0.743	0.804	0.2948	0.708	0.716	0.779	0.2976	**0.742**	**0.755**	**0.802**	**0.2946**
all	-	0.569	0.5	0.4905	0.862	0.862	0.928	0.2133	0.864	0.864	0.937	0.1846	**0.905**	**0.905**	**0.963**	**0.1625**
dir.	-	0.853	0.499	0.2513	0.945	0.946	0.949	0.0915	0.941	0.943	0.955	0.0933	**0.950**	**0.951**	**0.962**	**0.0886**
non.	-	0.593	0.5	0.4829	0.853	0.853	0.923	0.2329	0.813	0.814	0.893	0.2371	**0.878**	**0.879**	**0.941**	**0.2075**

Note: Column“prec.” represents “precision" (as shown in Section 4.3.5). Column “err." stands for “Mean absolute error" (as shown in Section 4.3.5). To make clear which model performs better than the others, we highlight the values of the model achieving the best performance among the four in **bold**, that of second best in underline. The 7th row marked with “all” means that the results are based on the overall database (among all projects). The 8th row marked with “dir.” means that the results are based on the methods with direct tests (among all projects). The 9th row marked with “non.” means that the results are based on the methods with no direct tests (among all projects).

Summary *The correlations between code observability metrics and mutation score are not very strong (<0.5); however, they are significantly better than the correlations for existing code metrics. Test directness (`test_distance` and `direct_test_no.`) takes the first place of `NSUP(class)` in |`rho`| among all metrics (including existing ones in Section 4.2.2), followed by assertion-based metrics (`assertion-density`, `assertion-McCabe` and `assertion_no`).*

4.4.2. RANDOM FOREST

Classification effectiveness As discussed in Section 4.3.4, we compare the four models in terms of both our code observability metrics and the existing metrics, namely:

1. `ZeroR`: model using ZeroR approach

Table 4.10: Feature importance of classification model among different projects (1)

1		2		3		4		5	
metric	imp.	metric	imp.	metric	imp.	metric	imp.	metric	imp.
test_distance	0.35	test_distance	0.15	test_distance	0.13	test_distance	0.48	test_distance	0.23
NLOC(class)	0.15	HIER(class)	0.12	NOCL	0.05	method_length	0.03	is_void	0.1
NOCL	0.03	CCML(class)	0.05	HDIF	0.03	COMP	0.03	EXCT	0.04
CREF	0.03	NLOC(class)	0.05	MI(class)	0.03	NOCL	0.03	NOCL	0.03
MINC(class)	0.03	NOCL	0.04	is_static	0.02	CAST	0.03	NOS	0.03
non_void_percent	0.02	MI(class)	0.04	non_void_percent	0.02	HDIF	0.03	S-R(class)	0.03
HDIF	0.02	assertion-density	0.03	HVOC	0.02	(cond)	0.02	is_public	0.02
NOS(class)	0.02	CREF	0.03	HEFF	0.02	VREF	0.02	nested_depth	0.02
PACK(class)	0.02	HDIF	0.03	CREF	0.02	is_void	0.01	direct_test_no	0.02
TCC(class)	0.02	PACK(class)	0.03	VREF	0.02	direct_test_no	0.01	assertion_no	0.02
LMC(class)	0.02	method_length	0.02	NEXP	0.02	assertion_no	0.01	CREF	0.02
HLTH	0.01	HVOC	0.02	HEFF(class)	0.02	non_void_percent	0.01	HDIF	0.02
HVOC	0.01	HEFF	0.02	PACK(class)	0.02	assertion-density	0.01	PACK(class)	0.02
HEFF	0.01	LMET	0.02	CBO(class)	0.02	HLTH	0.01	F-IN(class)	0.02
XMET	0.01	NOA	0.02	CCML(class)	0.02	HVOC	0.01	method_length	0.01

Note: Column “imp." denotes “feature importance" mentioned in 4.3.4.

Table 4.11: Feature importances of classification model among different projects (2)

6		all		dir.		non-dir.	
metric	imp.	metric	imp.	metric	imp.	metric	imp.
CBO(class)	0.09	test_distance	0.29	is_void	0.22	test_distance	0.16
HDIF	0.07	PACK(class)	0.06	PACK(class)	0.13	NOCL	0.09
NQU(class)	0.06	NOCL	0.05	HDIF	0.05	non_void_percent	0.04
test_distance	0.04	is_void	0.03	NOS	0.04	EXCT	0.04
non_void_percent	0.03	EXCT	0.03	assertion-density	0.03	HDIF	0.03
HVOC	0.03	non_void_percent	0.02	NEXP	0.03	PACK(class)	0.03
HEFF	0.03	CREF	0.02	direct_test_no	0.02	MI(class)	0.03
CREF	0.03	HDIF	0.02	assertion_no	0.02	CREF	0.02
XMET	0.03	MI(class)	0.02	assertion-McCabe	0.02	CBO(class)	0.02
NAND	0.03	is_public	0.01	NOCL	0.02	MINC(class)	0.02
VREF	0.03	is_nested	0.01	CREF	0.02	HIER(class)	0.02
NOA	0.03	method_length	0.01	NOA	0.02	F-IN(class)	0.02
NEXP	0.03	nested_depth	0.01	MINC(class)	0.02	MOD(class)	0.02
method_length	0.02	assertion_no	0.01	method_length	0.01	is_public	0.01
NOCL	0.02	getter_percent	0.01	nested_depth	0.01	is_static	0.01

Note: Column “imp." denotes “feature importance" mentioned in 4.3.4. Column “all” means that the results are based on the overall database (among all projects). Column “dir.” means that the results are based on the methods with direct tests (among all projects). Column “non-dir.” means that the results are based on the methods with no direct tests (among all projects)

2. `existing`: Random Forest model based on existing code metrics
3. `code observability`: Random Forest model based on code observability metrics
4. `combined`: Random Forest model based on the combination of existing metrics and code observability metrics.

The comparison of the four models is shown in Table 4.9. To make clear which model performs better than the others, we highlighted the values of the model achieving the best performance among the four in **bold**, that of second best in underline. For precision, recall, and AUC, the model with the best performance is the one with the highest value, while for the mean absolute error, the best scoring model exhibits the lowest value. For the ZeroR model, because this model classifies all the instances to the majority (i.e., one class), the precision of the minority is not valid due to 0/0. Thus, in Table 4.9, we mark the precisions by “-".

From Table 4.9, we can see that the Random Forest models are better than the baseline `ZeroR` which only relies on the majority. This is the *prerequisite* for further comparison. `Combined` achieves the best performance (in 5 out of 6 projects) compared to the existing code metrics and code observability metrics in terms of AUC; this observation is as expected since `combined` considered both the existing and our metrics during training, which provides the classification model with more information. The only exception is `java-apns-apns-0.2.3` (pid = 4). We conjecture that the number of instances (selected methods) in this project might be too small (only 150 methods) to develop a sound prediction model. In second place comes the model based on code observability metrics, edging out the model based on existing metrics.

For the overall dataset (the 7th row marked with “all” in Table 4.9), `combined` takes the first place in all evaluation metrics. In second place comes the `code observability`, slightly better than `existing`. Another angle which is interesting to investigate further is the *test directness*. If we only consider the methods that are directly tested (the second to last row in Table 4.9), `combined` again comes in first, followed by the existing code metrics model. The same observation holds for the methods that are not directly tested (the last row in Table 4.9). It is easy to understand that when the dataset only considers methods that are directly tested (or not), the test directness features in our model become irrelevant. However, we can see that the difference between existing metrics and ours are quite tiny (<3.4%).

Feature importance analysis Tables 4.10 and 4.11 show the top 15 features per project (and overall) in descending order. We can see that for five out of the six projects (including the overall dataset), `test_distance` ranks first. This again supports our previous findings that *test directness* plays a significant role in mutation testing. The remaining features in the top 14 vary per projects; this is not surprising, as the task and context of these projects vary greatly. For example, `Apache Commons Lang` (Column “2” in Table 4.10) is a utility library that provides a host of helper methods for the `java.lang API`. Therefore, most methods in `Apache Commons Lang` are public and static; thus, `is_public` and `is_static` are not among the top 15 features for `Apache Commons Lang`. A totally different context is provided by the `JFreeChart` project (Column “5” in Table 4.10). `JFreeChart` is a Java chart library, whose class encapsulation and inheritance hierarchy are well-designed, so `is_public` appears among the top 15 features.

Looking at the overall dataset (Column “all” in Table 4.11), there are eight metrics from our proposed code observability metrics among the top 15 features. The importance of `test_distance` is much higher than the other features (>4.83X). In second place comes `PACK(class)`, or the number of packages imported. This observation is easy to understand since `PACK(class)` denotes the complexity of dependency, and dependency could influence the difficulty of testing, especially when making use of mocking objects. Thereby, dependency affects the mutation score. Clearly, more investiga-

Table 4.12: Spearman results of test directness vs. assertions in terms of rho

rho	assertion_no	assertion-McCabe	assertion_distance
direct_test_no	0.9604	0.9472	0.9334
test_distance	-0.8707	-0.8707	-0.8707

Note: "rho" represents the pairwise correlation coefficient between each metric in test directness category and that in assertion category (as shown in Table 4.5).

tions are required to draw further conclusions. The third place in the feature importance analysis is taken by NOCL, which stands for the Number of Comments. This observation is quite interesting since NOCL is related to how hard it is to understand the code (*code readability*). This implies that code readability might have an impact on mutation testing.

As for the methods with direct tests (Column "dir." in Table 4.11), is_void takes the first position, which indicates that it is more difficult to achieve a high mutation score for void methods. Considering the methods without direct tests (Column "non-dir." in Table 4.11), test_distance again ranks first.

Another observation stems from the comparison of the performance of assertion related metrics in the feature importance analysis and the Spearman rank order correlation results (in Section 4.4.1). For Spearman's rank order correlation, we can see that assertion related metrics are the second significant category right after test directness (in Table 4.8 in Section 4.4.1). While in the feature importance analysis, assertion related metrics mostly rank after the top 5 (shown in Table 4.10 and Table 4.11). To further investigate the reason behind the dramatic changes of ranks for assertion related metrics, we analyse the correlations between test directness (i.e., direct_test_no and test_distance) and assertion related metrics (i.e., assertion_no, assertion-McCabe and assertion_distance). Looking at the correlation results between test directness and assertion related metrics in Table 4.12, the major reason is that test directness and assertion related metrics are almost *collinear* in the prediction model (where $|rho| > 0.87$). To put simply, there are almost no tests without assertions for the six subjects. If the method has a direct test, then the corresponding assertion no. is always greater than 1. Therefore, the ranks of assertion related metrics are not as high as we had initially expected in the feature importance analysis.

Moreover, we would like to put our observations into perspective by comparing our results with the work of Zhang et al. [378], where they have constructed a similar Random Forest model to predict killable mutant result based on a series of features related to mutants and tests. The metrics that are common to their model and ours are Cyclomatic Complexity (COMP), Depth of Inheritance Tree (DIT), nested_depth, Number of Subclasses (NSUB), and method_length. Only two metrics in their study, i.e., method_length (in 6th place) and nested_depth (in 10th place) appear in our top 15 (Column "all" in Table 4.11). Especially COMP which ranks nine in their results is not in our top 15. There are multiple reasons for the difference in results: (i) we do consider a much larger range of metrics, which provide a better explanatory power (statistically speaking) than the one in their paper; (ii) our goal is to determine patterns in production and test code that may prevent killing some mutants while Zhang et al. [378] predict if

Table 4.13: Selected features by PCA

is_public	(cond)	assertion-density	XMET
is_static	(cond(cond))	COMP	LMET
is_void	(cond(loop))	NOCL	NLOC
is_nested	(loop)	NOS	VDEC
method_length	(loop(cond))	HLTH	TDN
nested_depth	(loop(loop))	HVOC	NAND
direct_test_no	non-void_percent	HEFF	LOOP
test_distance	getter_percent	HBUG	MOD
assertion_no	assertion-McCabe	CREF	NOPR

a mutant is killable (aka different prediction target and different granularity level). Besides, as we see later (next section), we can use our model to determine common anti-patterns with proper statistical methods. (iii) the subjects used in our experiment are different from theirs. For example, in project `java-apns-apns-0.2.3` (Column "4" in Table 4.11), `COMP` appears among the top 15.

Summary *Overall, Random Forest based on the combination of existing code metrics and code observability metrics perform best, followed by that on code observability metrics. The analysis of feature importances shows that* test directness *ranks highest, remarkably higher than the other metrics.*

4.5. RQ4.4 CODE REFACTORING

Our goal is to investigate whether we can refactor away the observability issue that we expect to hinder tests from killing mutants and thus to affect the mutation score. In an in-depth case study, we manually analyse 16 code fragments to understand better the interaction between testability/observability metrics that we have been investigating, and the possibilities for refactoring.

Our analysis starts from the `combined` model, which as Table 4.9 shows, takes the leading position among the models. We then apply *Principal Component Analysis (PCA)* [366] to perform feature selection, which, as Table 4.13 shows, leaves us with 36 features (or metrics). Then, as discussed in Section 4.3, we build a decision tree based on those 36 metrics using J48 (shown in Figure 4.4), and select the top 6 leaves (also called end nodes) in the decision tree for further *manual* analysis as potential refactoring guidelines. We present the top six anti-patterns in Table 4.14.

Here, we take a partial decision tree to demonstrate how we generate rules (shown in Figure 4.5). In Figure 4.5, we can see that there are three attributes (marked as an ellipse) and four end nodes or leaves (marked as a rectangle) in the decision tree. Since we would like to investigate how code refactoring increases mutation score (**RQ4.4**), we only consider the end nodes labeled with "LOW" denoting mutation score<0.5. By combining the conditions along the paths of the decision tree, we obtain the two rules for "LOW" end nodes (as shown in the first column of the table in Figure 4.5). For every end node, there are two values attached to the class: the first is the number of instances that correctly fall into the node, the other is the instances that incorrectly fall into the node. The accuracy in the table is computed by the number of correct instances divided by that of total instances. As mentioned earlier, we select the top 6 end nodes from the

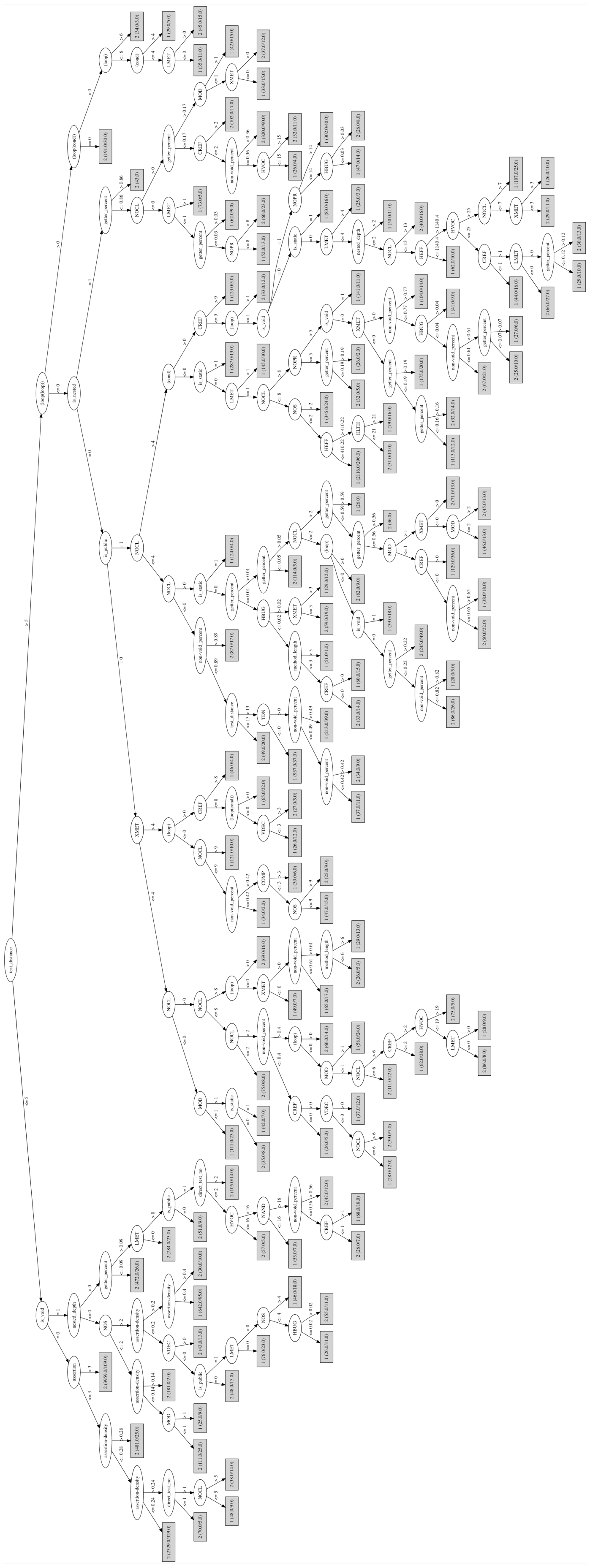
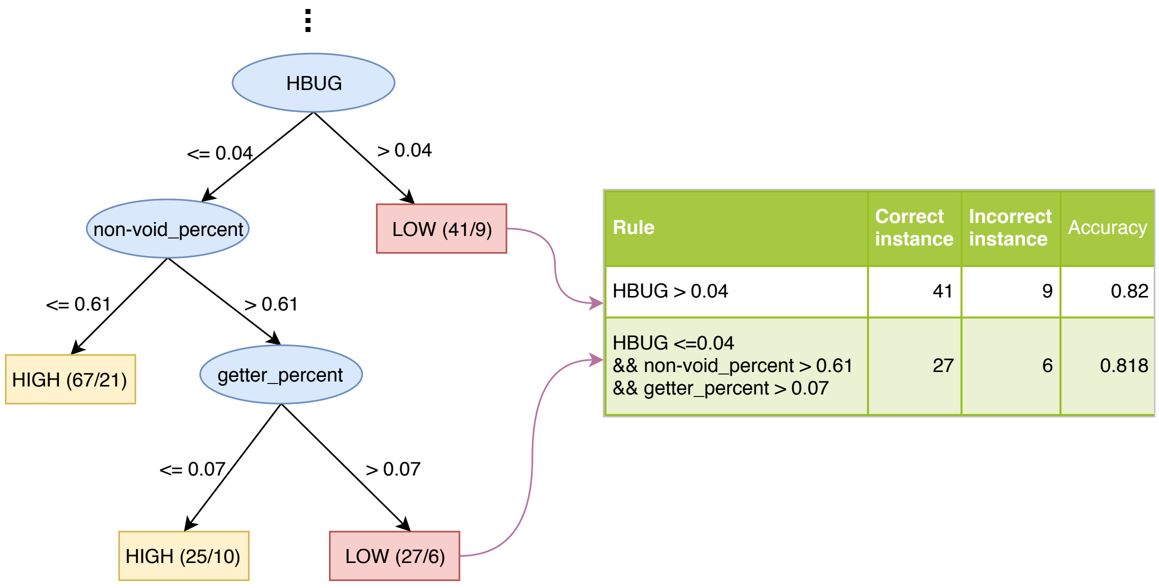


Figure 4.4: Overview of J48 decision tree

Table 4.14: Top six anti-patterns from J48 decision tree 4.4

Rule No.	Details
1	`test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) ≤ 0 && NOCL ≤ 9 && non-void_percent ≤ 0.42`
2	`test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) ≤ 0 && NOCL > 9`
3	`test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 1 && NOCL ≤ 4 && NOCL > 0 && is_static = 0 && getter_percent ≤ 0.01 && HBUG ≤ 0.02 && method_length > 3`
4	`test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 1 && NOCL > 4 && (cond) ≤ 0 && is_static = 0 && LMET ≤ 1 && NOCL > 8 && NOPR > 5 && is_void = 1`
5	`test_distance ≤ 5 && is_void = 1 && nested_depth ≤ 0 && NOS ≤ 2 && assertion-density ≤ 0.14 && MOD > 1`
6	`test_distance ≤ 5 && is_void = 1 && nested_depth ≤ 0 && NOS > 2 && assertion-density ≤ 0.22 && CREF > 1 && XMET > 0`



Rule	Correct instance	Incorrect instance	Accuracy
HBUG > 0.04	41	9	0.82
HBUG <=0.04 && non-void_percent > 0.61 && getter_percent > 0.07	27	6	0.818

Figure 4.5: Demo of rule generation

decision tree, where the end nodes are ranked by the number of correct instances under the condition accuracy≥0.8.

After selecting the rules, the first author of this chapter has conducted the main task of the manual analysis. If there were any questions during the manual analysis, the attempts of refactoring or adding tests are discussed among all the authors to reach an agreement. In our *actual* case study, we manually analyse 16 cases in total. Due to space limitations, we only highlight six cases in this chapter (all details are available on GitHub [390]). We will discuss our findings in code refactoring case by case.

Table 4.15: Summary of mutants from Listing 4.4 (Case 1)

ID	Line No.	Mutator	Results
1	1146	removed call to java/awt/Graphics2D::setFont	SURVIVED
2	1147	removed call to java/awt/Graphics2D::setPaint	SURVIVED
3	1149	negated conditional	SURVIVED
4	1151	negated conditional	SURVIVED
5	1157	Replaced float addition with subtraction	SURVIVED

```
1139  /**
1140   * Draws the value label just below the center of the dial.
1141   *
1142   * @param g2 the graphics device.
1143   * @param area the plot area.
1144   */
1145  protected void drawValueLabel(Graphics2D g2, Rectangle2D area) {
1146    g2.setFont(this.valueFont);
1147    g2.setPaint(this.valuePaint);
1148    String valueStr = "No value";
1149    if (this.dataset != null) {
1150      Number n = this.dataset.getValue();
1151      if (n != null) {
1152        valueStr = this.tickLabelFormat.format(n.doubleValue()) + " "
1153            + this.units;
1154      }
1155    }
1156    float x = (float) area.getCenterX();
1157    float y = (float) area.getCenterY() + DEFAULT_CIRCLE_SIZE;
1158    TextUtils.drawAlignedString(valueStr, g2, x, y,TextAnchor.TOP_CENTER);
1159  }
```

Listing 4.4: `plot.MeterPlot::drawValueLabel` (Case 1)

4.5.1. Case 1: plot.MeterPlot::drawValueLabel from *JFreeChart*

This case (shown in Listing 4.4) is under anti-pattern **Rule 1**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) ≤ 0 && NOCL ≤ 9 && non-void_percent ≤ 0.42`. In total, there are 5 mutants generated from this method (shown in Table 4.15). All 5 mutants survive the test suite.

Code changes We start with `test_distance > 5` which means there is no direct test for this method. Accordingly, we add one direct test (shown in Listing 4.5).

However, Mutant 4 and 5 cannot be killed by adding the above direct test. Upon inspection, we found that Mutant 4 and 5 cannot be killed because the `DrawValueLabel(...)` method is void. In particular, this means that the changes in the state caused by the `TextUtils.drawAlignedString()` method (line 1158) cannot be assessed. This is indicated by `non-void_percent ≤ 0.42` in **Rule 1**. We then refactor the method to have it return `Rectangle2D` (shown in Listing 4.6). Also, we improve the direct test for this method in Listing 4.5 by adding a new test method (shown in Listing 4.7) to avoid the *assertion roulette* test smell [254, 284]. By refactoring the method to non-void and adding a direct test, all previously surviving mutants are now successfully killed.

```java
@Test
public void testDrawValueLabel(){
  MeterPlot p1 = new MeterPlot(new DefaultValueDataset(1.23));
  BufferedImage image = new BufferedImage(3, 4, BufferedImage.TYPE_INT_ARGB)
      ;
  Graphics2D g2 = image.createGraphics();
  Rectangle2D area = new Rectangle(0, 0, 1, 1);
  p1.drawValueLabel(g2,area);
  assertTrue(g2.getFont() == p1.getValueFont());
  assertTrue(g2.getPaint() == p1.getValuePaint());
}
```

Listing 4.5: Direct test for Listing 4.4 (Case 1)

```java
protected Rectangle2D drawValueLabel(Graphics2D g2, Rectangle2D area) {
  g2.setFont(this.valueFont);
  g2.setPaint(this.valuePaint);
  String valueStr = "No value";
  if (this.dataset != null) {
    Number n = this.dataset.getValue();
    if (n != null) {
      valueStr = this.tickLabelFormat.format(n.doubleValue()) + " "
          + this.units;
    }
  }
  float x = (float) area.getCenterX();
  float y = (float) area.getCenterY() + DEFAULT_CIRCLE_SIZE;
  return TextUtils.drawAlignedString(valueStr, g2, x, y,TextAnchor.TOP_CENTER);
}
```

Listing 4.6: Refactoring of Listing 4.4 (Case 1)
Note: we highlight the refactored parts in yellow colour.

Table 4.16: Summary of mutants from Listing 4.8 (Case 2)

ID	Line No.	Mutator	Results
1	165	mutated return of Object value for org/jfree/chart/util/PaintAlpha::darker to (if (x != null) null else throw new RuntimeException)	NO_COVERAGE
2	166	Replaced double multiplication with division	NO_COVERAGE
3	167	Replaced double multiplication with division	NO_COVERAGE
4	168	Replaced double multiplication with division	NO_COVERAGE

4.5.2. CASE 2: AXIS.SYMBOLAXIS::DRAWGRIDBANDS FROM *JFreeChart*

This case (shown in Listing 4.8) is under **Rule 2**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 0 && XMET > 4 && (loop) ≤ 0 && NOCL > 9`. In total, 4 mutants are generated from this method (see Table 4.16). None of the mutants are killed.

Code changes It is clear that this method is private, thus, it is impossible to call this method from outside the class directly. We first refactor this method from `private` to `public`. This is revealed by `is_public = 0` in **Rule 2**.

Then, guided by `test_distance > 5` from **Rule 2**, we add a direct test for this method to kill all mutants (see Listing 4.10).

```
@Test
public void testDrawValueLabelArea() {
  MeterPlot p1 = new MeterPlot(new DefaultValueDataset(1.23));
  BufferedImage image = new BufferedImage(3, 4, BufferedImage.TYPE_INT_ARGB)
      ;
  Graphics2D g2 = image.createGraphics();
  Rectangle2D area = new Rectangle(0, 0, 1, 1);
  Rectangle2D drawArea = p1.drawValueLabel(g2,area);
  assertEquals(0.5,drawArea.getCenterX(),0.01);
  assertEquals(18.8671875,drawArea.getCenterY(),0.01);
  assertEquals(15.0,drawArea.getHeight(),0.01);
  assertEquals(64.0,drawArea.getWidth(),0.01);
}
```

Listing 4.7: Improved direct test for Listing 4.4 (Case 1)
Note: we highlight the improved parts in yellow colour.

```
154 /**
155  * Similar to {@link Color#darker()}.
156  * <p>
157  * The essential difference is that this method
158  * maintains the alpha-channel unchanged<br>
159  *
160  * @param paint a {@code Color}
161  *
162  * @return a darker version of the {@code Color}
163  */
164 private static Color darker(Color paint) {
165   return new Color(
166       (int)(paint.getRed () * FACTOR),
167       (int)(paint.getGreen() * FACTOR),
168       (int)(paint.getBlue () * FACTOR), paint.getAlpha());
169 }
```

Listing 4.8: `axis.SymbolAxis::drawGridBands` (Case 2)

4.5.3. CASE 3: `BUILDER.IDKEY::HASHCODE` FROM *Apache Commons Lang*

This case (shown in Listing 4.11) is under **Rule 3**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 1 && NOCL ≤ 4 && NOCL > 0 && is_static = 0 && getter_percent ≤ 0.01 && HBUG ≤ 0.02 && method_length > 3`. Only one mutant is generated for this method: a mutant that replaces the return value with (x == 0 ? 1 : 0). This mutant survives.

Code changes Starting with `test_distance > 5`, we add a direct test for this method (shown in Listing 4.12), which works perfectly to kill the mutant.

4.5.4. CASE 4: `ABSTRACTCATEGORYITEMRENDERER::DRAWOUTLINE` FROM *JFreeChart*

This case (shown in Listing 4.13) is under **Rule 4**: `test_distance > 5 && (loop(loop)) ≤ 0 && is_nested = 0 && is_public = 1 && NOCL > 4 && (cond) ≤ 0 && is_static`

```
public static Color darker(Color paint) {
  return new Color(
      (int)(paint.getRed () * FACTOR),
      (int)(paint.getGreen() * FACTOR),
      (int)(paint.getBlue () * FACTOR), paint.getAlpha());
}
```

Listing 4.9: Refactoring of Listing 4.8 (Case 2)
Note: we highlight the refactored parts in yellow colour.

```
@Test
public void testDarker(){
  Color paint = new Color(10,20,30);
  Color darker = PaintAlpha.darker(paint);
  assertEquals(7,darker.getRed());
  assertEquals(14,darker.getGreen());
  assertEquals(21,darker.getBlue());
}
```

Listing 4.10: Direct test for Listing 4.8 (Case 2)

`= 0 && LMET ≤ 1 && NOCL > 8 && NOPR > 5 && is_void = 1`. Also in this case, only 1 mutant is generated for this method. The particular change applied is the removal of the call to `AbstractCategoryPlot::drawOutline`. The original test suite did not kill the mutant.

Code changes Based on `test_distance > 5`, we add one direct test (as shown in Listing 4.14) for this method to kill the surviving mutant.

4.5.5. CASE 5: `builder.ToStringStyle::setUseShortClassName` FROM *Apache Commons Lang*

This case (shown in Listing 4.15) is under **Rule 5**: `test_distance ≤ 5 && is_void = 1 && nested_depth ≤ 0 && NOS ≤ 2 && assertion-density ≤ 0.14 && MOD > 1`. In this case, a single (surviving) mutant is generated that removes the call to `builder.ToStringStyle::setUseShortClassName`.

Code changes We can see that **Rule 5** is different from the previous rule in that `test_distance` is less than 5, while in **Rule 4** `test_distance > 5`. A more in-depth analysis reveals that

```
  /**
   * returns hash code - i.e., the system identity hashcode.
   * @return the hashcode
   */
  @Override
  public int hashCode() {
    return id;
  }
```

Listing 4.11: `builder.IDKey::hashCode` (Case 3)

```java
@Test
public void testHashCode(){
  IDKey idKey = new IDKey(new Integer(123));
  assertEquals(989794870,idKey.hashCode());
}
```

Listing 4.12: Direct test for Listing 4.11 (Case 3)

```java
/**
 * Draws an outline for the data area. The default implementation just
 * gets the plot to draw the outline, but some renderers will override this
 * behaviour.
 *
 * @param g2 the graphics device.
 * @param plot the plot.
 * @param dataArea the data area.
 */
@Override
public void drawOutline(Graphics2D g2, CategoryPlot plot,
    Rectangle2D dataArea) {
  plot.drawOutline(g2, dataArea);
}
```

Listing 4.13: `AbstractCategoryItemRenderer::drawOutline` (Case 4)

the method in Listing 4.15 is already directly invoked by the original test suite. The surviving mutant is due to the fact that there are no assertions that examine the changes after the `setUseShortClassName` method call. This situation is reflected by `assertion-density` $\le$ `0.14` in **Rule 5**. Therefore, we add assertions to assess the changes (seen in Listing 4.16), which leads to the mutant being killed.

4.5.6. CASE 6: EXCEPTION.TOOMANYEVALUATIONSEXCEPTION::<INIT> FROM *Apache Commons Math*

This case (shown in Listing 4.17) is under **Rule 6**: `test_distance` $\le$ `5 && is_void = 1 && nested_depth` $\le$ `0 && NOS > 2 && assertion-density` $\le$ `0.22 && CREF > 1 && XMET > 0 && VDEC` $\le$ `0 && NOCL` $\le$ `12`. A single mutant is generated: a removal of the call to `exception.util.ExceptionContext::addMessage`. This mutant is surviving the test suite.

```java
@Test
public void testDrawOutline(){
  AbstractCategoryItemRenderer r = new LineAndShapeRenderer();
  BufferedImage image = new BufferedImage(200 , 100,
      BufferedImage.TYPE_INT_RGB);
  Graphics2D g2 = image.createGraphics();
  CategoryPlot plot = new CategoryPlot();
  Rectangle2D dataArea = new Rectangle2D.Double();
  r.drawOutline(g2,plot,dataArea);
  assertTrue(g2.getStroke()==plot.getOutlineStroke());
}
```

Listing 4.14: Direct test for Listing 4.13 (Case 4)

```java
/**
 * <p>Sets whether to output short or long class names.</p>
 *
 * @param useShortClassName the new useShortClassName flag
 * @since 2.0
 */
@Override
public void setUseShortClassName(final boolean useShortClassName) { // NOPMD as this is implementing the abstract class
  super.setUseShortClassName(useShortClassName);
}
```

Listing 4.15: `builder.ToStringStyle::setUseShortClassName` (Case 5)

```java
@Test
public void testSetUseShortClassName(){
  assertTrue(STYLE.isUseShortClassName());
  STYLE.setUseShortClassName(false);
  assertFalse(STYLE.isUseShortClassName());
  STYLE.setUseShortClassName(true);
  assertTrue(STYLE.isUseShortClassName());
}
```

Listing 4.16: Additional assertions for Listing 4.15 (Case 5)

Code changes We found that the mutant in Line 37 cannot be killed because the function `addMessage` changes the field `List<Localizable> msgPatterns`. This field is private in the class `ExceptionContext` and there is no other way to access it. As such, our first step is to add a getter for `msgPatterns` (shown in Listing 4.18). In **Rule 6**, we can see that `is_void = 1` is the underlying cause since void methods could be difficult to test if no getters for private fields exist.

To kill the surviving mutant, we add one extra assertion (in a new test method) to examine the changes in `msgPatterns` (in Listing 4.19). This action is also partly evidenced by `assertion-density` ≤ `0.22` in **Rule 6**. As assertion-density denotes the ratio between the total number of assertions in direct tests and the lines of code in direct tests, low assertion-density is a sign of insufficient assertions in the direct tests to detect the mutant.

```java
30  /**
31   * Construct the exception.
32   *
33   * @param max Maximum number of evaluations.
34   */
35  public TooManyEvaluationsException(Number max) {
36    super(max);
37    getContext().addMessage(LocalizedFormats.EVALUATIONS);
38  }
```

Listing 4.17: `exception.TooManyEvaluationsException::<init>` (Case 6)

```
public List<Localizable> getMsgPatterns(){
  return msgPatterns;
}
```

Listing 4.18: Refactoring of Listing 4.17 (Case 6)

```
@Test
public void testMsgPatterns() {
  final int max = 12345;
  final TooManyEvaluationsException e = new TooManyEvaluationsException(max)
      ;
  final String msg = e.getLocalizedMessage();
  Assert.assertTrue(e.getContext().getMsgPatterns()
          .contains(LocalizedFormats.EVALUATIONS));
}
```

Listing 4.19: Additional assertion for Listing 4.17 (Case 6)
Note: we highlight the added parts in yellow colour.

4.5.7. RQ4.4 Summary

Based on all 16 cases that we analysed (available in our GitHub repository [390]), we found that our code observability metrics can lead to simple refactorings that enable to kill mutants that were previously not being killed. Ultimately, this leads to an increase of the mutation score:

- most cases can be easily fixed by adding direct tests if `test_distance>5`.
- most cases can be easily fixed by adding assertions if `test_distance≤5`.
- private methods must be refactored to protected/public for testing (indicated by `is_public=0`).
- three void methods had to be refactored to be non-void (indicated by `is_void=1` and `non-void_percent≤0.42`).
- one void method needed an additional getter because a private field was changed (indicated by `is_void=1`).

4.5.8. Discussion

From the findings of **RQ4.4**, we can see that some code refactorings break OO design principles [81]. For instance, we suggest to change the access modifier from *private* to *protected/public* to kill the mutants; this violates the idea of Encapsulation, the ability to protect some components of the object from external entities [81]. This observation brings us to a discussion about the dilemma between OO design principles and testing and hence software testability [335]. The main concepts of OO design are centred around the features of Data abstraction, Encapsulation, Inheritance, Polymorphism, and Dynamic binding. However, some factors such as Encapsulation and Inheritance could increase the complexity of OO systems and hence hinder testing and testability [335]. Existing literature [257, 263, 323, 385] has already addressed this dilemma. Mouchawrab et al. [257] pointed out that increasing the size of the inheritance hierarchy could increase the cost of testing due to dynamic dependencies. Singh and Saha's work [323] has shown that Inheritance and Polymorphism increase testing effort and lower software testability. All the works above indicate that there is a trade-off between OO design features and software testability. Currently, it is up to practitioners to balance the two perspectives themselves depending on the requirements of software and their preferences.

In the context of mutation testing, a similar trade-off between OO design features and the ease of killing mutants exists. In this study, we relate the ease of killing mutants to the testability and observability. In Section 4.5.7, we found that a simple strategy to kill *all* the mutants is to write additional direct tests and/or assertions. However, some OO design features related to Encapsulation, such as the *private* access modifier (see Listing 4.8), increase the difficulty to add a direct test. Also, the void return type prevents killing the mutants generated from the immediate states that cannot propagate to the output (see Listing 4.4). As such, a very important note here is that our refactoring recommendations listed in Section 4.5.7 are centred around the anti-patterns based on the testability and observability; they *do not* take OO design principles into consideration. The recommendations attempt to help developers in understanding the cause of the low mutation score considering testability and observability, but not all surviving mutants are due to test quality.

Take Listing 4.8 for instance. The developer found the mutation score of this method is low, and our tool shows the low mutation score is mainly due to *private* access control modifier. Then, the developer can decide to ignore the surviving mutants if he cannot break Encapsulation based on the requirement. Or if this method is critical and must be well-tested according to the document, he may alter the access control modifier from *private* to *protected/public* to kill the mutants. Whether the developers make use of these testability and observability recommendations depends on their choices with regard to either (1) adding test cases [71–73] (2) refactoring the production code to kill the mutants, or (3) ignoring the surviving mutants.

4.6. THREATS TO VALIDITY

External validity Our results are based on mutants generated by the operators implemented in PIT. While PIT is a frequently used mutation testing tool, our results might be different when using other mutation tools [220]. Concerning the subject systems selection, we choose six open-source projects from GitHub; the selected projects differ in size, the number of test cases and application domain. Besides, as mentioned in Section 4.3, the large number of the methods with low mutant number is an unavoidable bias in our experiment. The reason is partly due to the optimisation mechanism of PIT [101] and partly due to a large number of short methods in those projects. Nevertheless, we do acknowledge that a broad replication of our study would mitigate any generalisability concerns even further.

Internal validity The main threat to internal validity for our study is the implementation of the Mutation Observer tool for the experiment. To reduce internal threats to a large extent, we rely on existing tools that have been widely used, e.g., `WEKA`, `MATLAB`, and PIT. Moreover, we carefully reviewed and tested all code for our study to eliminate potential faults in our implementation. Another threat to internal validity is the disregard of equivalent mutants in our experiment. However, this threat is unavoidable and shared by other studies on mutation testing that attempt to detect equivalent mutants or not [165, 252]. Moreover, we consider equivalent mutants as a potential weakness in the software (reported by Coles [100, slide 44-52]); thereby, we did not manually detect equivalent mutants in this chapter.

Construct validity The main threat to construct validity is the measurement we used to evaluate our methods. We minimise this risk by adopting evaluation metrics that are widely used in research (such as recall, precision, and AUC), as well as a sound statistical analysis to assess the significance (Spearman's rank-order correlation).

4.7. Related work

The notion of *software testability* dates back to 1991 when Freedman [151] formally defined *observability* and *controllability* in the domain of software. Voas [354] proposed a dynamic technique coined propagation, infection, and execution (PIE) analysis for statistically estimating the program's *fault sensitivity*. More recently, researchers have aimed to increase our collective understanding of *testability* by using statistical methods to predict *testability* based on various code metrics. A prime example is the work of Bruntink and van Deursen [89], who have explored the relationship between nine class-level object-oriented metrics and testability. To the best of our knowledge, no study uses statistical or machine learning methods to investigate the relationship between *testability*/*observability* metrics and the mutation score.

Mutation testing was initially introduced as a fault-based testing method which was regarded as significantly better at detecting errors than the *covering measure* approach [92]. Since then, mutation testing has been actively investigated and studied, thereby resulting in remarkable advances in its concepts, theory, technology, and empirical evidence. For more literature on mutation testing, we refer to the existing surveys of DeMillo [115], Offutt and Untch [277], Jia and Harman [197], Offutt [279] and our literature review (Chapter 2). Here we mainly address the studies that concern *mutant utility* [203], the efficacy of mutation testing. Yao et al. [373] have reported on the causes and prevalence of equivalent mutants and their relationship to stubborn mutants based on a manual analysis of 1230 mutants. Visser [351] has conducted an exhaustive analysis of all possible test inputs to determine how hard it is to kill a mutant considering three common mutation operators (i.e., relational, integer constants and arithmetic operators). His results show that *mutant reachability, mutation operators and oracle sensitivity* are the key contributions to determining how hard it is to kill a mutant. Just et al. [203] have shown a strong correlation between mutant utility and context information from the program in which the mutant is embedded. Brown et al. [88] have developed a method for creating potential faults that are more closely coupled with changes made by actual programmers where they named “wild-caught mutants". Chekam et al. [96] have investigated the problem of selecting the fault revealing mutants. They put forward a machine learning approach (decision trees) that learns to select fault revealing mutants from a set of static program features. Jimenez et al. [198] investigated the use of natural language modeling techniques in mutation testing. All studies above have enriched the understanding of mutation testing, especially its efficacy. However, the aim of our work is different from those studies, as we would like to gain insights into how code quality in terms of testability and observability affects the efforts needed for mutation testing, especially in how to engineer tests to kill *more* the mutants.

Similar to our study, there have been a few recent studies also investigating the relationships between assertions and test directness with mutation testing. Schuler and Zeller [315] introduced *checked coverage*—the ratio of statements that contribute to the computation of values that are later checked by the test suite— as an indicator for oracle quality. In their experiment, they compared checked coverage with the mutation score, where they found that checked coverage is more sensitive than mutation testing in evaluating oracle quality. Huo and Clause [187] proposed *direct coverage and indirect coverage* by leveraging the concepts of *test directness* with conventional statement coverage. They used the mutants as an indicator of the test suite effectiveness, and they found faults in indirectly covered code are significantly less likely to be detected than those in directly covered code. Zhang and Mesbah [384] evaluated the relationship between test suite effectiveness (in terms of the mutation score) and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions. They found test assertion quantity and assertion coverage are strongly correlated with the mutation score, and assertion types could also influence test suite effectiveness. Compared to our studies, those works only addressed one or two aspect(s) of *code observability* in our study. We provide a complete view of the relationships between code observability and mutation testing.

The study most related to ours is that of Zhang et al. [378]’s *predictive mutation testing*, where they have constructed a classification model to predict killable mutant result based on a series of features related to mutants and tests. In their discussion, they compared source code related features and test code related features in the prediction model for the mutation score. They found that test code features are more important than source code ones. But from their results, we cannot draw clear conclusions on the impact of production code on mutation testing as their goal is to predict exact killable mutant results. Another interesting work close to our study is Vera-Pérez et al. [347]’s *pseudo-tested methods*. Pseudo-tested methods denote those methods that are covered by the test suite, but for which no test case fails even if the entire method body is completely stripped. They rely on the idea of “extreme mutation”, which completely strips out the body of a method. The difference between Vera-Pérez et al. [347]’s study and ours is that we pay attention to *conventional* mutation operators rather than “extreme mutation”.

4.8. Conclusion & Future Work

This chapter aims to bring a new perspective to software developers helping them to understand and reason about the mutation score in the light of *testability* and *observability*. This should enable developers to make decisions on the possible actions to take when confronted with low mutation scores. To achieve this goal, we firstly investigate the relationship between *testability* and *observability* metrics and the mutation score. More specifically, we have collected 64 existing source code quality metrics for testability, and have proposed a set of metrics that specifically target *observability*. The results from our empirical study involving 6 open-source projects show that the 64 existing code quality metrics are not strongly correlated with the mutation score ($|rho| < 0.27$). In contrast, the 19 newly proposed *code observability metrics*, that are defined in terms of both production code and test cases, do show a stronger correlation with the mutation score ($|rho| < 0.5$). In particular, *test directness*, `test_distance`, and `direct_test_no` stand out.

To better understand the causality of our insights, we continue our investigation with a manual analysis of 16 methods that scored particularly bad in terms of mutation score, i.e., a number of mutants were not killed by the existing tests. In particular, we have refactored these methods and/or added tests according to the anti-patterns that we established in terms of the code observability metrics. Our aim here is to establish whether the removal of the observability anti-patterns would lead to an increase in the mutation score. We found that these anti-patterns can indeed provide insights in order to kill the mutants by indicating whether the production code or the test suite needs improvements. For instance, we found that private methods (expressed as `is_public=0` in our schema) are prime candidates to potentially refactor to increase their observability, e.g., by making them public or protected for testing purpose.

However, some refactoring recommendations could violate OO design principles. For example, by changing *private* to *protected/public* we increase observability, but we also break the idea of encapsulation. Therefore, we suggest developers make a choice between—(1) adding test cases, (2) refactoring the production code to kill the mutants, or (3) ignoring the surviving mutants—by considering the trade-off between OO design features and testability/observability.

To sum up, our chapter makes the following contributions:

1. 19 newly proposed *code observability* metrics
2. a detailed investigation of the relationship between *testability*/*observability* metrics and the mutation score (**RQ4.1-RQ4.3**)
3. a case study with 16 code fragments to investigate whether removal of the anti-patterns increases the mutation score (**RQ4.4**)
4. a guideline for developers to make choices when confronting low mutation scores
5. a prototype tool coined MUTATION OBSERVER (openly available on GitHub [390]) that automatically calculates code observability metrics

Future work With our tool, and since the results are encouraging, we envision the following future work: 1) conduct additional empirical studies on more subject systems; 2) evaluate the usability of our code observability metrics by involving practitioners; 3) investigate the relations between more code metrics (e.g., code readability) and mutation score.

5 MUTATION TESTING FOR PHYSICAL COMPUTING

Physical computing, which builds interactive systems between the physical world and computers, has been widely used in a wide variety of domains and applications, e.g., the Internet of Things (IoT). Although physical computing has witnessed enormous realisations, testing these physical computing systems still face many challenges, such as potential circuit related bugs which are not part of the software problems, the timing issue which decreasing the testability, etc.; therefore, we proposed a mutation testing approach for physical computing systems to enable engineers to judge the quality of their tests in a more accurate way. The main focus is the communication between the software and peripherals. More particular, we first defined a set of mutation operators based on the common communication errors between the software and peripherals that could happen in the software. We conducted a preliminary experiment on nine physical computing projects based on the Raspberry Pi and Arduino platforms. The results show that our mutation testing method can assess the test suite quality effectively in terms of weakness and inadequacy.

This chapter has been published in the 18th IEEE International Conference on Software Quality, Reliability, and Security (QRS) [394].

5.1. Introduction

Physical computing creates a conversation between the physical world and the virtual world of the computer [282]. The recent confluence of embedded and real-time systems with wireless, sensor, and networking technologies is creating a nascent infrastructure for an educational, technical, economic, and social revolution. Fuelled by the recent adoption of a variety of enabling wireless technologies such as RFID tags, embedded sensor and actuator nodes, the Internet of Things (IoT) has stepped out of its infancy and is rapidly advancing in terms of technology, functionality, and size, with more real-time applications [166]. A good example of the IoT is wearable devices like fitness trackers that are ever getting more popular.

Modern embedded platforms, like those centred around the 8051 and Freescale microcontroller series, have seen a dramatic rise in speed and functionality. The Raspberry Pi and Arduino platforms, which were originally meant for education, are two of the most popular modern embedded platforms. They are both open-source electronics platforms based on easy-to-use hardware and software.

An equally important trend is *softwarization of hardware*. In the early days, hardware engineers had to build circuits by physically connecting electronic components using wire and soldering. More recently, *reconfigurable computing* tools provide the opportunity to compile programs written in high-level languages such as C and Java into a hardware architecture. A Raspberry Pi supports several programming languages including Python to control the General Purpose Input/output (GPIO) pins to communicate with the external devices. This means that developing a physical computing system has been simplified to the point where the hardware peripherals can easily be controlled via software without even knowing the hardware part. This trend also provides a great opportunity for applying methodologies of software engineering in physical computing, especially testing techniques.

As physical computing is maturing, testing these sensor-based applications, especially the processing programs, becomes essential. Essential, because compared to conventional software projects, the costs associated with failing physical computing systems are often even bigger, as bugs can result in real-life accidents. For example, a robotic arm might accidentally hurt the human if the programmer does not set up the initial state properly. Therefore, to develop a rigorous and sound physical computing system, a high-quality test suite becomes crucial. This brings us to *mutation testing*, a fault-based testing technique that assesses the test suite quality by systematically introducing small artificial faults [197]. It has been shown to perform well in exposing faults [144, 228, 244].

In this chapter, we propose a novel mutation testing approach for physical computing systems enabling engineers to judge the quality of their tests in an accurate way. Specifically, we define a set of mutation operators based on *common mistakes* that we observed when developing physical computing systems. We present an initial evaluation of our approach on the Raspberry Pi and Arduino platforms.

5.2. Background and motivation

We introduce basic concepts related to *physical computing* and *mutation testing*. We then motivate why mutation testing should be applied to physical computing systems.

5.2.1. PHYSICAL COMPUTING

Most physical computing systems (and most computer applications in general) can be broken down into the following same three stages: input, processing, and output [282]. The input is about how computers sense the physical world via sensors and signals, such as buttons and speakers. While the output is where computers make changes to the world under people's desire through various actuators, like servos, motors and LEDs. The processing procedure requires a computer (usually an embedded platform) to read the inputs and turn them into outputs.

The ***General Purpose Input/output (GPIO)*** is the primary interface that micro-controllers including Raspberry Pi use to communicate with external devices. The pins available on a processor can be programmed to be used to either accept input or provide output to external devices depending on user desires and application requirements. These pins support a variety of data handling methods, such as Analog-to-Digital conversion and interrupt handling. GPIO is also the main focus of our methodology.

Among different embedded system platforms, the ***Raspberry Pi*** is a popular one-chip computer which includes an ARM-compatible CPU, a GPU and a Secure Digital (SD) card module. Its recommended operating system for normal use is Raspbian, a free, Debian-based operating system optimised for the platform. Python is the recommended programming language and the *RPi.GPIO* library is used to configure GPIO pins.

Another popular micro-controller is ***Arduino*** which is also open-source and easy-to-use. Arduino boards support GPIO pins as well. The Arduino Software (IDE) runs on the Windows, MacOS, and Linux operating systems. Its programming language can be expanded through C++ libraries, and users wanting to understand the technical details can make the leap from Arduino to the AVR C programming language on which it is based. Similarly, users can also add AVR-C code directly into Arduino programs.

5.2.2. MUTATION TESTING

Mutation testing is defined by Jia and Harman [197] as a fault-based testing technique which provides a testing criterion called the *mutation adequacy score*. This score can be used to measure the effectiveness of a test set regarding its ability to detect faults [197]. The principle of mutation testing is to introduce syntactic changes into the original program to generate faulty versions (called *mutants*) according to well-defined rules (mutation operators) [279]. The benefits of mutation testing have been extensively investigated and can be summarised as (mentioned in Chapter 2): 1) having better fault exposing capability compared to other test coverage criteria [144, 228, 244], 2) being an excellent alternative to real faults and providing a good indication of the fault detection ability of a test suite [50].

5.2.3. CHARACTERISTICS OF PHYSICAL COMPUTING

Physical computing allows to build interactive physical systems through a combination of hardware and software. The following six major characteristics describe the uniqueness of physical computing [329]:

(1) Safety and security issue: physical computing systems are much more safety-critical than traditional software where small defects could have a tremendous impact on the reliability of systems upon which people's lives and living depend. Moreover, sensor networks interact closely with their physical environment and with people, posing additional security problems.

(2) Fault-tolerance: fault-tolerance is a crucial requirement for physical computing systems that manage to handle exceptions properly once a certain part does not work. E.g., sensors may fail due to surrounding physical conditions or when their energy runs out. It may be difficult to replace existing sensors; the network must be fault-tolerant such that non-catastrophic failures are hidden from the application [339].

(3) Lack of knowledge: physical computing is a multi-dis-ciplinary domain which requires developers to create high-level software system as well as low-level embedded systems solutions. However, most embedded systems developers have an electrical engineering background, therefore, might lack basic knowledge of software engineering, especially testing techniques; which could lead to error-prone code and low-quality tests.

(4) Circuit related bugs: this type of errors is mostly due to hardware configuration, such as shorts circuits, errors in sensors, undefined states (not pulling up resistors for the input processing); these bugs could be prevented or localised by testing each component at the unit level.

(5) Timing issue: in most cases, peripherals are activated or deactivated at a particular time, e.g., systems embedded with sonic sensors only start working when the distance meets a specific condition. The timing issue decreases testability of physical computing systems as it is hard to set up the real scenarios for testing.

(6) Slow execution speed: although there is a dramatic improvement in the power and functionality of modern embedded platforms, the execution speed of these embedded platforms is still not as comparable as PCs and servers. Thereby, the processing program must be carefully designed to avoid computationally-consuming algorithms.

Motivation. We can see that physical computing systems require extremely error-free and reliable code considering the *safety and security issue* and *fault-tolerance*. The *slow execution speed* of embedded platforms also demands a well-designed and cost-effective processing program to be deployed ubiquitously. Moreover, the testing procedure is of utmost importance to implement high-quality and error-free programs, as well as detect *circuit related bugs,* and make up developers' *lack of knowledge* of software engineering. Also, a weak test suite is not sufficient enough to detect the faults and cannot correctly handle the *timing issue*.

Taking all the characteristics of physical computing systems together, the primary challenge for physical computing systems here is: *how to effectively and efficiently test these physical systems?* To deal with this challenge, we are seeking to apply software engineering methodologies to the physical computing domain. In particular, mutation testing, which is well-known for its high fault-revealing effectiveness, is a viable way to help developers design better quality test suites in this highly safety critical domain. Also, mutation testing, as a fault injection technique, is an ideal method for testing the fault tolerance mechanisms with respect to a specific set of inputs the physical computing systems are meant to cope with [58].

5.3. DESIGNING MUTATION OPERATORS

To integrate computing with the physical world via sensors and actuators, an essential component is an interface between the software (processing programs) and peripherals (sensors and actuators). The proliferation of sensor and actuator networks in (civilian) applications requires new approaches to handle real-time, multimedia and multi-threaded communications, such as wireless sensor network [339] and cloud computing [125]. This leads to a more complex and error-prone integration part. Therefore, when designing a mutation operator for physical computing, our main goal is to narrow down the scope of the mutation process to parts of the code that affect the communication between the software and peripherals (digital circuits[1]), namely the GPIO interface. To derive the mutation operators that represents errors typically made by programmers during the implementation of the software, we first summarise *common mistakes* that could happen in the software based on our experience. Subsequently, we design a set of mutation operators for these common mistakes.

(1) output value errors: The output value is usually decided by a complex function which takes many elements such as feedback from the sensors and preferences of the user, into consideration. For example, an automatic watering system decides when to water the plants according to multiple environmental conditions, e.g., soil humidity and the amount of water configured by the user. Thus, the output could be wrong if there exists a bug in the function. More specifically, we only pay attention to the final output value generated by the function, i.e., whether the output value is high or low, regardless of the function details. For this type of the error, we derived the *OutputValueReplacement (OVR)* operator which replaces *HIGH* to *LOW* (and vice versa) in the output value.

(2) output setting omissions: Once a certain signal has been received/read by a peripheral, the output value should, in some cases, be reset to ensure that the peripheral can change states at a later stage. For example, a self-driving car should reduce engine output when detecting a wall, but the engine should engage again after clearing the wall. Accordingly, we designed the *OutputSettingRemoval (OSR)* operator which deletes the output setting function.

(3) pin number errors: The programmer may read information or send control signals using a wrong pin that she does not intend to operate. The problem typically arises during prototyping for two reasons: 1) the GPIO pins are usually on the PCB as a symmetric array without labels so that designers need to locate a pin by counting, and 2) the order of a pin on the PCB is typically different from its numerical ID in the software API, making the mapping error-prone. *PinNumberReplacement (PNR)* replaces the pin id with one of the surrounding pin ids.

(4) input value errors: There are usually two ways to obtain an input value. The simplest way is to check the input value at a point in time. This “polling" can potentially miss an input if the program reads the value at the wrong time. The other way of responding to a GPIO input is using edge detection. An edge is the name of a transition from HIGH to LOW (falling edge) or LOW to HIGH (rising edge). Quite often, we are more concerned by a change in state of an input than its value. One potential fault in the edge detection is to mix up the falling and rising edge. This problem is common due to the confusion brought by the variety of external devices, e.g., for the 7400 series logic chip, for instance, the 74LS107 JK flip-flop chip [338] triggers on a rising edge, while the 74HC74 D flip-flop chip [337] triggers on a falling edge. For input value mistakes, we defined the following

[1]In this chapter, we focus on the digital circuits, where two possible states, i.e., *HIGH* and *LOW*, are considered. As this is the fundamental circuit type compared to *analog*.

Table 5.1: Summary of mutation operators

Mutation operator	Full name	Definition
OVR	Output Value Replacement	replace *HIGH* to *LOW* (and vice versa) in the output value
OSR	Output Setting Removal	delete the output setting function
PNR	Pin Number Replacement	replace the pin id with its surrounded pin ids
IVR	Input Value Replacement	replace *HIGH* to *LOW* (and vice versa) in the input value
EDR	Edge Detection Replacement	replace edge names among {*FALLING*, *RISING*, *BOTH*}
IOMR	I/O Mode Replacement	replace *IN* to *OUT* (and vice versa) in the mode setting
SIR	Setup Input Replacement	replace the input value from *PUD_UP* to *PUD_DOWN* (and vice versa) in setup function
SOR	Setup Output Replacement	replace the output value from *HIGH* to *LOW* (and vice versa) in setup function
SVR	Setup Value Removal	remove the initial value setting in setup function for both input and output modes

two mutation operators:

- *InputValueReplacement (IVR)*: replaces *HIGH* to *LOW* (and vice versa) in the input value
- *EdgeDetectionReplacement (EDR)*: replaces *FALLING* to *RISING* (and vice versa) in the edge detection. However, sometimes, there is one more edge event called *BOTH* which covers both the falling and rising edge. In this case, the replacement happens among the three edge events, e.g., replace *FALLING* to *RISING* and *BOTH*.

(5) I/O pin mode errors: A GPIO pin allows to define each individual pin on the chip as being in *input* or *output* mode. As a side-effect of pin number mistakes, the programmer might set the pin I/O mode by mistake. Thus we designed the *I/OModeReplacement (IOMR)* operator that changes *IN* to *OUT* (or vice versa).

(6) initial setup value errors: If a pin is not "connected" to a peripheral, it will "float". In other words, the value that is read in is undefined because it is not connected to anything. It could frequently change values as a result of receiving mains interference. To get around this, GPIO modules usually provide an option to use a pull-up (*PUD_UP*) or pull-down (*PUD_DOWN*) resistor to set the default value of the input. Two potential errors in this context are (1) the omission of setting up the input value or (2) initializing it with the opposite value by mistake. Similarly, for the output mode, pins can have different default output values in a single GPIO module. The initial output value affects the initial state of the peripheral that the pin is connected to, which could lead to a breakdown or unexpected activation. For instance, if the pin connected to a motor is initially set to *HIGH*, then once the module is activated, the motor is immediately activated which is supposed to be activated when the switch is on. The potential errors in output value setup are similar to the input value setup, i.e., the setup omission and the initial value mistakes. Accordingly, there are three mutation operators:

- *SetupInputReplacement (SIR)*: replace the input value from *PUD_UP* to *PUD_DOWN* (and vice versa) in setup function.

- *SetupOutputReplacement (SOR)*: replace the output value from *HIGH* to *LOW* (and vice versa) in setup function.

- *SetupValueRemoval (SVR)*: remove the initial value in setup function for both input and output modes.

Summary: We designed nine mutation operators (summarised in Table 5.1) to replicate common communication errors in physical computing systems.

5.4. TOOL IMPLEMENTATION

Various modern embedded platforms contain the GPIO module, such as Arduino, BeagleBone, PSoC kits and Raspberry Pi. In this chapter, we chose Raspberry Pi and Arduino as the target platforms to implement the aforementioned mutation operators. One thing to note is that our approach should work with the other aforementioned platforms as well.

We have coined our mutation tool MUTPHY and implemented it in Python. The overall architecture of MUTPHY is shown in Figure 6.4. MUTPHY consists of two components, i.e., the mutation engine and the test executor. MUTPHY takes the program and its test suite as input. First, the mutation engine analyses the source code and marks all possible mutation points, and then the mutation generator produces all the mutants according to mutation operators. After that, the program and generated mutants together with the test suite go to the test executor where the mutation testing is performed: each mutant is executed against the test suite one by one. Finally, MUTPHY prints out the detailed mutant killable results. The main task of the code analyser is to analyse the test dependencies and parse the source code of the program for the mutation generator. The mutation generator contains all the mutation operators and the details of the mutants including the mutation location (line number) and the mutation operator type.

As Raspberry Pi and Arduino are the target platforms, we have created two variants of MUTPHY. The main differences between the two variants are inherent to the programming languages that are supported by two platforms. For Raspberry Pi, the code analyser of MUTPHY needs to parse Python, as Python is Raspberry Pi's recommended programming language. As Arduino only supports C/C++, we created a C/C++ code analyser in MUTPHY for Arduino. Moreover, we considered *pytest* [27], a non-boilerplate alternative to Python's standard *unittest* testing framework [40], as the test executor for both the Raspberry Pi and Arduino platforms , as it can also handle other popular Python testing libraries, e.g. *unittest* and *doctests* [7].

5.5. EMPIRICAL EVALUATION

To assess the efficacy of our mutation testing approach, we conducted an experimental study using two embedded system platforms, i.e. Raspberry Pi and Arduino. We proposed the following research questions to steer our experimental study:

- **RQ5.1:** *How effective is* MUTPHY *in evaluating the existing test suite?* With this research question, we evaluate to what extent MUTPHY can effectively evaluate the quality of the existing test suite.

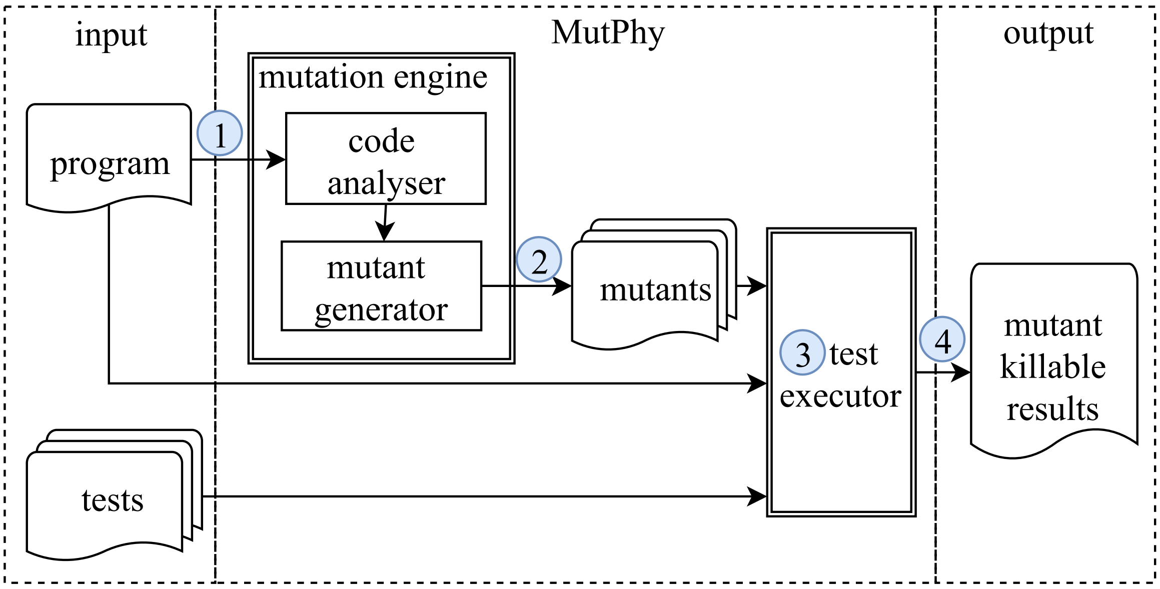


Figure 5.1: Overview of MutPhy architecture and workflow

- **RQ5.2:** *How efficient is* MutPhy *in generating non-equivalent mutants?* As we designed the mutation operators based on common mistakes made by programmers, this might lead to potential *redundant* mutation operators which are subsumed by others. **RQ5.2** addresses the efficiency of MutPhy in generating non-equivalent mutants.

- **RQ5.3:** *Is it possible to kill all non-equivalent surviving mutants by adding extra test cases?* This research question focuses on non-equivalent surviving mutants and aims to assess whether our approach enables engineers to write a better test suite.

For **RQ5.1**, we determined the effectiveness of our approach based on the number of non-equivalent surviving mutants. Also, we compared our results to test coverage. To answer **RQ5.2**, we manually analysed the generated mutants to determine whether the mutant is equivalent to the original program. For **RQ5.3**, we analysed the non-equivalent surviving mutants in detail and tried to manually engineer new test cases to kill these mutants.

5.5.1. Case Studies with Raspberry Pi

In the first part of the experiment, we use five Raspberry Pi based projects for evaluating MutPhy. For these five projects, four are obtained from GitHub, and one is from industry (Guangzhou Kompline Electronics). The four open source projects have been manually selected from GitHub under the Raspberry Pi topic using the following process: we (1) sorted by stars (from high to low), (2) checked whether they contain “GPIO" as a keyword, (3) verified that they are implemented in Python, and (4) examined whether they can be successfully built, and (5) inspected whether they contain a test suite. Since our main focus is the GPIO interface, we only apply mutation operators on the files that use

Table 5.2: Subjects based on Raspberry Pi

Project	File	LOC	#Tests	Coverage
RPLCD	gpio.py	99	35	71%
hcsr04sensor	sensor.py	93	6	96%
jean-pierre	buzzer.py	20	21	41%
gpiozero	mock.py	312	302	97%
four-wheel robot	arm.py	179	11	93%
	chassis.py	158	4	100%
Total		861	379	82.8%

Note: Column "LOC" standing for the line of code is measured by sloccount[363]. Column "Coverage" means statement coverage measured by Coverage.py[5].

the GPIO library. Table 5.2 summarises the main characteristics of the selected projects.

When answering the **RQ**s in the next sections, we will start with **RQ5.2**, as we need to analyse non-equivalent mutants to calculate the mutation score which is part of **RQ5.1**.

RPLCD

The project *RPLCD* [66] is a Python 2/3 Raspberry PI Character LCD library for the Hitachi HD44780 [16] controller. The main peripheral of this system is a LCD module.

Table 5.3: Mutants result of *RPLCD*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	13	10	13	0	0	0
OSR	11	10	11	0	1	0
PNR	47	41	47	0	0	0
IVR	0	0	0	0	0	-
EDR	0	0	0	0	0	-
IOMR	1	1	1	0	0	0
SIR	0	0	0	0	0	-
SOR	0	0	0	0	0	-
SVR	0	0	0	0	0	-
Overall	72	62	72	0	1	0

Note: Column "MOP" means mutation operator. Column "#Equiv." means the number of the equivalent mutants. Column "MS" means nutation score.

Using MUTPHY, we generated 72 mutants for the *RPLCD* project. This project mainly uses the *GPIO.output* method to write data to the LCD board, thus, only four types of mutation operators can be applied to the system: OVR, OSR, PNR and IOMR. The details of all generated mutants are presented in Table 5.3. We can see from Table 5.3 that only one equivalent mutant is generated by MUTPHY. This equivalent OSR mutant is located in a statement that, under the existing test configuration, cannot be reached. Thus, for this LCD controlling system, the efficiency of MutPhy in generating non-equivalent mutants is promising (**RQ5.2**).

While the statement coverage is 71%, the mutation score is zero. Furthermore, 86.1% of mutants are covered by the test suite, but none of the mutants is actually killed. Why then is the mutation score of this project so low? We found that the developers replaced the *RPi.GPIO* module of the system under test with mock objects; this allows the tests to be executed without a Raspberry Pi. As a side effect, the developers did not assess the communication between the software and peripherals for this system. The above findings indicate that compared to statement coverage, the mutation score can better represent how a test suite examines the behaviour of GPIO pins (**RQ5.1**).

To kill the mutants (**RQ5.3**), we first removed the mock objects for the *RPi.GPIO* module and executed the test suite on an actual Raspberry Pi. This modification led to 21 PNR and 1 IOMR mutants killed. Then, we analysed whether the remaining mutated statements are covered by the tests or not. As shown in Table 5.3, we found 85.2% non-equivalent mutants to be covered by the test suite. However, the existing test suite only calls the functions in *gpio.py* file, but does not check the behaviour of the GPIO pins. To address this drawback of the existing test suites, we added five test cases to examine all the pins once their states changed. To capture the state change sequence of GPIO pins, we introduced new mock objects. Different from the system developers' mock objects, we used mock objects to increase the observability of the system under test. For instance, one method in *gpio.py* file called *pulse_enable()*, that sends a pulse signal to tell the LCD board to process the data. The method *pulse_enable()* calls *GPIO.output* three times in one pin generating a LOW-HIGH-LOW signal. Without a mock object of method *GPIO.output*, it is hard to tell what happens to this pin after this function call, as the starting and the ending states are both LOW. With the additional five test cases, all the non-equivalent mutants are killed.

hcsr04sensor

The *hcsr04sensor* project [60] is a Python module for measuring distance and depth with a Raspberry Pi and HC-SR04 Ultrasonic Module [16], which uses sonar to determine the distance to an object, just like bats or dolphins do. The sensor first emits ultrasound at 40,000 Hz, which travels through the air and if there is an object or obstacle on its path, the ultrasound will bounce back to the module. Considering the travel time and the speed of the sound, it calculates the distance. The HC-SR04 Ultrasonic Module has 4 pins: Ground, VCC, Trig and Echo.

Table 5.4 details the generated mutants for *hcsr04sensor*. In total, MutPhy generated 41 mutants. For this system, the Raspberry Pi controls the HC-SR04 Ultrasonic Module by writing to the Trig pin and reading from Echo. As such, this control program mainly adopts *GPIO.output* and *GPIO.input* methods. This results in five types of mutants from OVR, OSR, PNR, IVR and IOMR operators. There are no equivalent mutants generated by our proposed mutation operators; this indicates MutPhy has high efficiency in generating non-equivalent mutants (**RQ5.2**).

For **RQ5.1**, although 100% of the mutants are covered, 22% of the mutants are *not* detected by the test suite. Looking at the existing test suite, we found that the test suite checked all the initial settings of each GPIO pins, but lacks tests to (1) examine the pins'

Table 5.4: Mutants result of *hcsr04sensor*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	3	3	1	2	0	0.67
OSR	3	3	1	2	0	0.67
PNR	31	31	7	24	0	0.77
IVR	2	2	0	2	0	1
EDR	0	0	0	0	0	-
IOMR	2	2	0	2	0	1
SIR	0	0	0	0	0	-
SOR	0	0	0	0	0	-
SVR	0	0	0	0	0	-
Overall	41	41	9	32	0	0.78

Note: Column “MOP" means mutation operator. Column “#Equiv." means the number of the equivalent mutants. Column “MS" means nutation score.

state changes during the execution and (2) the final states after tearing down. For this project, it is important to clean up the Trig and Echo pins after use, because otherwise the distance cannot be accurately calculated by a new request to the ultrasonic sensor. This gives another indication that mutation score is a better metric of test suite quality than statement coverage, which only reveals insufficient tests for the system.

Regarding **RQ5.3**, we observe seven PNR mutants that are still alive; all originating from the *GPIO.cleanup* function. To kill these mutants, we need to add two additional assertions at the point just after the pins are torn down which means the pins are not used anymore. Once the pins are torn down, they cannot be read from or written to anymore, so the assertions expect exceptions when trying to read those pins.

The other two alive mutants, one of type OVR and one other of type OSR, are located on the same line, more precisely when calling the *GPIO.output* function. Similar to the *pulse_enable()* method in project *RPLCD*, this *GPIO.output* function is meant to send a LOW value, the first stage of the pulse signal. We follow a similar strategy in that we try to introduce mock objects to increase the observability, but this modification led to a syntax error: a local variable *sonar_signal_on* is referenced before assignment. Through further investigation, we found that this local variable is only assigned right after the Echo pin detects a HIGH signal via the *GPIO.input* function, while in the situation with mocks, the *GPIO.input* function is not actually invoked. This leaves us in the situation that if we do not introduce mock objects, the state change of this *GPIO.output* function cannot be observed, while if we do introduce mock objects, there is a syntax error.

The aforementioned observation is a case of a *snarled method*, a term coined by Feathers to describe a method dominated by a single large, indented section [138]. Feathers suggest to perform an *extract method* refactoring to move all the statements related to the pulse signal into a separate method [138]. In doing so, we create a function *pulse_enable()* and we separate responsibilities of this *snarled method*. As a result, we can easily test the state change caused by the target *GPIO.output* function without affecting the remaining part. For these two mutants, it is hard to derive new tests to kill them without refactoring the original production code. Through refactoring, the statement where the mutants are located is moved from a long method to a short one, thus, improving the observability of the state change made by the statement. This raises an interesting speculation: the testability of the production code [254] could have an influence on the test suite's mutation score. In Voas et al.'s work [355], they proposed that software testability could be defined for different types of testing, such as data-flow testing and mutation testing. Their work inspires us to explore the relationship of software testability and mutation testing in the future work.

jean-pierre

The project *jean-pierre* [94] is a little DIY robot based on the Raspberry Pi Zero W [29]. It uses a camera to scan food barcodes: it fetches information about the product from the OpenFoodFacts API [25] and adds it to a grocery list that the user can manage from a web interface. Once an object is successfully added to the grocery list, a buzzer makes two beeps. This system consists of three components: a Raspberry Pi Zero W, a Raspberry Pi Camera Module [28] and a buzzer. The main use of the GPIO pins in this project is to control the buzzer (*buzzer.py* file).

Table 5.5: Mutants result of *jean-pierre*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	2	0	2	0	0	0
OSR	2	0	2	0	0	0
PNR	6	0	6	0	0	0
IVR	0	0	0	0	0	-
EDR	0	0	0	0	0	-
IOMR	1	0	1	0	0	0
SIR	0	0	0	0	0	-
SOR	0	0	0	0	0	-
SVR	0	0	0	0	0	-
Overall	11	0	11	0	0	0

Note: Column "MOP" means mutation operator. Column "#Equiv." means the number of the equivalent mutants. Column "MS" means nutation score.

As the buzzer only has one function, i.e., *beep()*, it mainly adopts the *GPIO.output* function. When running our tool, 11 mutants are generated (shown in Table 5.5). For **RQ5.2**, no equivalent mutant is generated, which shows MUTPHY's high efficiency in generating non-equivalent mutants. For **RQ5.1**, we can see that the mutation score is 0 while the statement coverage is 41%. Although the statement coverage is 41%, none of the generated mutants is covered by the test suite. Closer inspection revealed that there are no tests in the existing test suite that are specifically designed to test the communication of the software and the buzzer. We can see that the mutation score enables to evaluate how the test suite examines the integration part of the software and peripherals in physical computing systems, while the test coverage cannot.

To kill the mutants (**RQ5.3**), we first added a test case to cover the mutants without assertions. Once the mutated statements are covered, i.e., the statement coverage reaches 100%, the six alive PNR and one alive IOMR mutants are killed. These seven mutants can easily be detected once the mutated GPIO pins are invoked, because the

RPi.GPIO module throws exceptions if these pins are either not initialised or initialised incorrectly. For instance, GPIO8 pin is called without initialisation, or GPIO9 pin is written to HIGH after being initialised to input mode. Then, to kill the remaining four surviving mutants, we again introduced mock objects to assess each state change made by the *GPIO.output* function. By designing effective test oracles to test the state change of the GPIO pins using mock objects, all the mutants are killed.

gpiozero

The project *gpiozero* [308] is a simple interface to GPIO devices with Raspberry Pi, which requires minimal boilerplate code to get started. This project is developed by the Raspberry Pi Foundation. This library provides many simple and obvious interfaces for the essential components, such as LED, Button, Buzzer, sensors, motors and even a few simple add-on boards.

Table 5.6: Mutants result of *gpiozero*

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	2	2	0	2	0	1
OSR	1	1	0	1	0	1
PNR	68	68	0	68	0	1
IVR	6	6	0	6	0	1
EDR	19	19	0	19	0	1
IOMR	14	14	0	14	0	1
SIR	8	8	0	8	0	1
SOR	2	2	0	2	0	1
SVR	5	5	1	4	1	1
Overall	125	125	1	124	1	1

Note: Column "MOP" means mutation operator. Column "#Equiv." means the number of the equivalent mutants. Column "MS" means nutation score.

Table 5.6 shows the 125 mutants generated by MUTPHY. For **RQ5.2**, there is only one equivalent mutant generated by MUTPHY. This equivalent mutant of type SVR stems from the initial value being removed from the setup function, yet with the default output value being the same as the initial value, there is an equivalence. For **RQ5.1**, the mutation score of this project is 1, which shows the existing test suite is adequate to detect all the mutants. One necessary condition for such a high mutation score is high test coverage. We can see that the statement coverage of the existing test suite is 97% and all the mutated statements are covered by the test suite. Moreover, there are 302 test cases in the existing test suite. Looking at the tests in detail, we found that each test case not only examines the basic information of the pin under test, i.e., the pin number and the pin state, but also other possible settings of the pin, e.g., I/O pin mode and resistor state. As the mutation score of this project has already achieved 1, there is no need for us to add extra tests to enhance the test quality (**RQ5.3**). From project *gpiozero*, we can conclude that the test suite can indeed achieve 100% mutation score when the GPIO pins are taken into consideration in tests and test oracles are carefully designed.

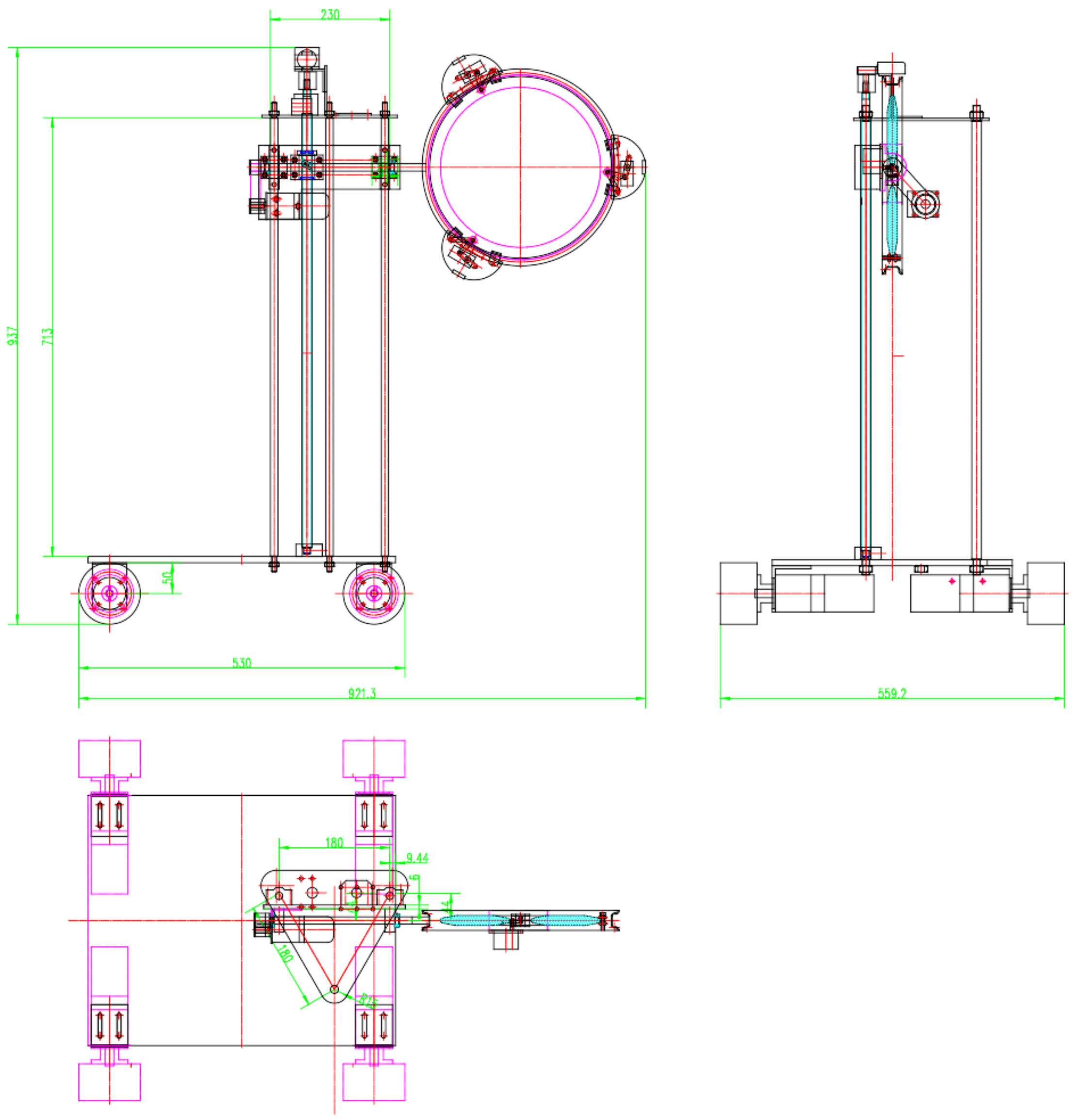


Figure 5.2: Three-view diagrams of four-wheel robot

four-wheel robot

This subject is a four-wheel robot, which has been designed and developed for industrial use (as shown in Figure 5.2). The robot is capable of moving pie-shaped objects from one place to another. During the movement, the robot may optionally rotate the object by at most 2π rad, and the four wheels can move it in six directions (as presented in Figure 5.3). The robot includes one Raspberry Pi 2, five photoelectric sensors, two DC motors, four stepper motors and three servos. The photoelectric sensors are mainly used to align the robot in specific positions (e.g., the starting point and the destination) based on differently coloured regions. The four stepper motors are responsible for the movement of the four wheels. As for the two DC motors, one drives the vertical movement of the robotic arm; the other is for the rotation of the arm. The three servos are used to control the action of the claw to grab the pie-shaped objects. The control system of

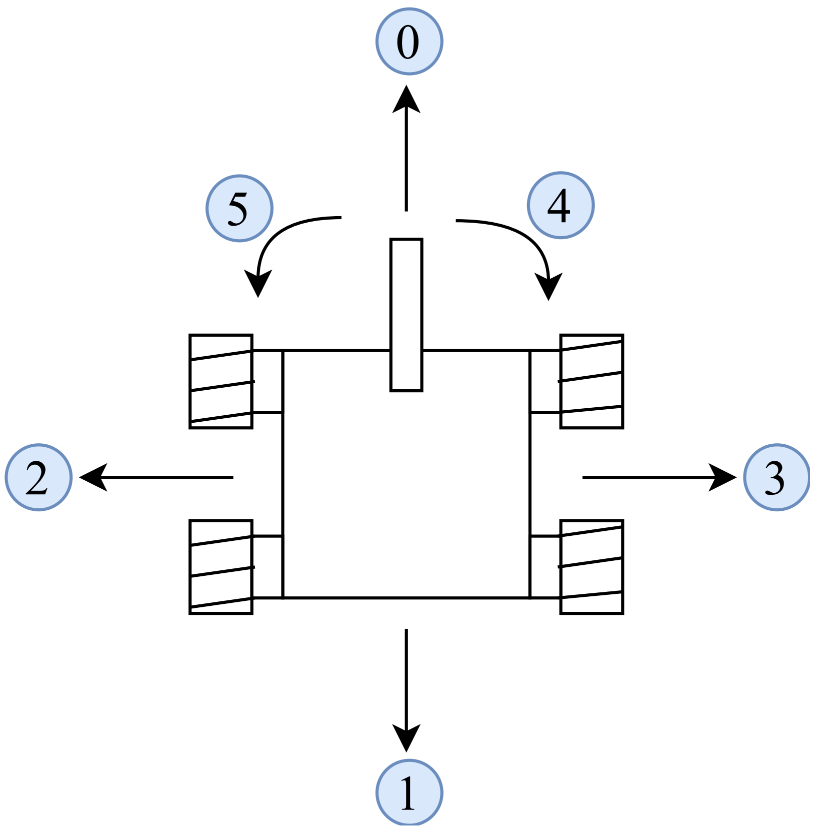


Figure 5.3: Movement directions of four-wheel robot

the robot consists of two parts, the chassis (*chassis.py*) and the arm (*arm.py*). The chassis part has 13 functions, and the arm part consists of 13 functions. The entire system's footprint comprises 337 lines of code. To set up a safe environment for testing, there is one test track with black and white lines designed for the robots. All the test cases are based on this test track. The test suite for the four-wheel robot system consists of 15 test cases totalling 243 lines of code. The statement coverage of the test suite is 96.5%.

Using MUTPHY, we generated 371 mutants. The summarised result of all generated mutants is presented in Table 5.7. For **RQ5.2**, we found there are 10 equivalent mutants generated by MUTPHY. Similar to project *gpiozero*, all the equivalent mutants are of type SVR, where the initial value assignment in the setup function is removed. The cause of the equivalence is also similar: the initial default value is the same as the explicitly set initial value. Although these mutants are equivalent to the original program, explicitly setting the initial value in the setup function is still recommended because different embedded platforms have different default values and setting the initial value can avoid unexpected initial states. In conclusion, for the four-wheel robot system, the efficiency of MUTPHY in generating non-equivalent mutants is high (97.3%).

For **RQ5.1**, the overall mutation score is 0.83, which is lower than the statement coverage (96.5%). The three mutation operators with the highest mutation score are IOMR (1), PNR (0.91) and IVR (0.81). The first two mutation operators are easier to be killed than the others because these mutants can be detected once the mutated GPIO pins are

Table 5.7: Mutants result of four-wheel robot

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	32	32	10	22	0	0.69
OSR	32	32	12	20	0	0.63
PNR	235	235	20	215	0	0.91
IVR	21	20	4	17	0	0.81
EDR	0	0	0	0	0	-
IOMR	19	19	0	19	0	1
SIR	3	3	3	0	0	0
SOR	13	13	7	6	0	0.46
SVR	16	16	15	1	10	0.17
Overall	371	370	71	300	10	0.83

Note: Column "MOP" means mutation operator. Column "#Equiv." means the number of the equivalent mutants. Column "MS" means nutation score.

invoked: in most cases, these pins are not initialised or initialised correctly (e.g., replace the output mode to the input mode). The 20 alive mutants from PNR are because of insufficient assertions in the tests suite; these missing assertions are needed to check the mutated statements. For IVR, as the input pins of the robots are connected to photoelectric sensors that are used to align the robot, most IVR mutants are easily killed if the robot does not reach the specific position by reading the un-mutated photoelectric sensors' states. For the four alive IVR mutants, one is due to uncovered statements; the other three are due to poor test design.

The three mutation operators with lowest mutation score are SIR (0), SVR (0.08) and SOR (0.46). The reason why none of the SIR mutants is killed is that the corresponding pins are connected to the peripherals (in particular, the photoelectric sensors) with very high resistors; this means the replacement of initial input value (*PUD_UP* or *PUD_DOWN*) cannot affect the overall potential. These alive mutants cannot be killed in this case, and even adding new tests would not make a difference. For SVR, the five alive non-equivalent mutants are due to insufficient assertions of the tests suite: the existing test suite does not examine all the initial states of the GPIO pins. The low mutation score of the SOR operator is due to inadequate tests that do not examine the initial states of the GPIO pins once the program starts.

The mutation score of mutants generated from OVR and OSR are 0.69 and 0.63, which is lower than we expected. The alive mutants of these two operators are due to meaningless feedback produced by the control program, and the test oracles are based on these feedback messages. For instance, the function *lift()* in *arm.py* lifts the arm for a given direction (up or down) and a period. Once the *lift()* call is finished, the function returns the input direction. This kind of feedback does not reflect the actual states of the GPIO pins. Thus, the corresponding tests can never fail. To kill these surviving mutants, we replaced *GPIO.output* functions with mock objects to assess intermediate states of the target pins. For the five mutants that are located in the method *lift()*, introducing mock objects enables to effectively detect these mutants. However, the 17 other mutants cannot be easily killed by making use of mock functions. These 17 mutants reside in complicated methods with loops and input detections. Similar to project *hcsr04sensor*, the intermediate changes cannot be easily captured and observed by introducing mock objects, as the sequence of the method calls is uncertain (another case of a *snarled method* [138]). Thus, we need to refactor the original control program by moving the related *GPIO.output* function calls into new methods; this enabled us to design accurate test oracles to examine the state changes.

For **RQ5.3**, we managed to kill the 51 non-equivalent alive mutants by adding and improving test cases. The remaining 20 non-equivalent surviving mutants cannot be killed by simply adding tests. Among the 20 mutants, 17 mutants can be killed by refactoring the production code. This observation strengthens our earlier assumption that the mutation score could be influenced by the testability of the production code. The other three non-killable SIR mutants are caused by the peripherals. More precisely, for the affected circuits the overall potential cannot be changed by pulling up or down resistor, as the resistor of peripherals is too high to be changed by the Raspberry Pi's function. This type of stubborn mutants is unique to physical computing systems when compared to conventional software; it also increases the difficulty of testing physical computing systems. We suggest to classify this type of stubborn mutants as equivalent mutants, as the peripherals are part of the system, and generally, this part is not likely to change once the system is built up.

5.5.2. CASE STUDIES WITH ARDUINO

The second part of our experiment targets the Arduino platform. The Arduino based system is taken from a lab session of an Embedded Software course for second-year undergraduate students at Delft University of Technology. The system is a robot that uses a camera instead of light or IR sensors to follow a line. It is shown in Figure 5.4 and is composed of of three components, each with a different role:

1. Smartphone: the camera of the smartphone is mounted on the robot makes images of the floor in front of the robot where the line should be detected;

2. Laptop: the laptop runs the Robot Operating System (ROS) core [281] and performs line detection on the images of the smartphone;

3. Arduino-based robot: the robot has to follow the line on the ground. This part includes one LCHB-100 H-bridge [20], one Arduino Mega ADK [55], one HC-05 Bluetooth dongle and one HC-SR04 ultrasonic sensor.

The students are required to implement the control program for the Arduino board and the line detection program based on ROS in groups of two. We collected implementations from four groups (the average LOC is 122.5 measured by sloccount[363]), and then the teaching assistant was asked to design test suites for those implementations. The main purpose of the test suites are to examine the five behaviours of the robot, i.e., going straight, turning left, turning right, stopping when there is an obstacle in front and stopping when no image is received. However, since the implementations of different groups are different from each other, we have to adjust the details of the tests to make them pass for further mutation testing. The statement coverage of the test suite is 100%.

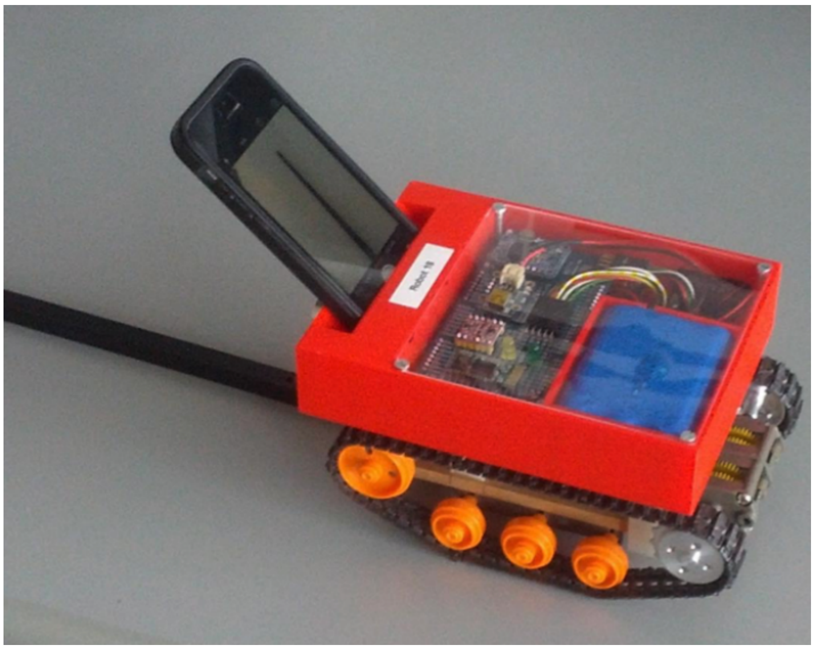


Figure 5.4: Diagram of line-follower robot

TEST ENVIRONMENT

The testing system is expected to be as isolated as possible from the program under test. In particular, the testing system should monitor the GPIO signals while keeping the code untouched. However, since the requirements of the student codes do not include the testing part, most of them cannot be tested without altering the codes. The reasons are as follows: first of all, the Arduino platform does not support multi-process nor multi-threading and thus only allows one main loop during execution. For the line-follower robot, the Arduino control program needs to be running continuously to receive operation signals from the PC as a client. Secondly, the test execution should be independent of the control program as a second process. In order to not introduce another process, we have to alter the students' code by adding test cases in the same program. This results in modifications and uncertainties in the control program. Therefore, we worked around the software limitation by adding a hardware monitor as shown in Figure 5.5. More specifically, we used another Arduino board (Arduino Uno [56]) to monitor the pin states of the control board of the line-follower robot.

The hardware monitor picks up two types of signals from the system under test.

- **Pulse Width Modulation (PWM) signals for the two DC motors for the wheels**. Each DC motor occupies a pair of PWM channels for the two rotational directions (controlled via LCHB-100 H-bridge). Therefore, the two DC motors take four PWM channels in total. We program the monitor hardware to sample the signals from the four channels at regular intervals. Thus, we can know whether the signal is high or low at each interval. We then approximate the duty cycle by calculating the ratio between the number of high signals and that of all signals. For instance, there are 100 high signals out of 500 detected in five seconds for one PWM channel. Thus, the approximated duty cycle is 20%.

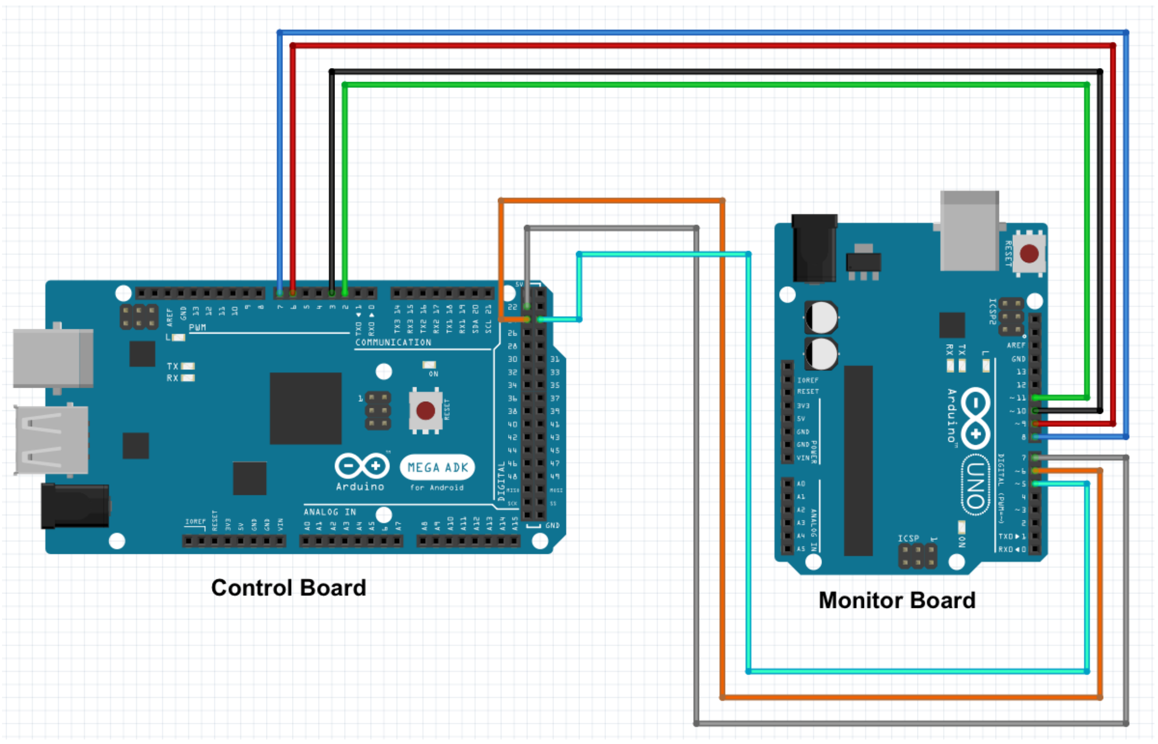


Figure 5.5: Layout of test setup of line-follower robot

- **Standard digital signals from the ultrasonic distance sensor**. The sensor (HC-SR04) has a trigger pin and an echo pin. The trigger pin is used to emit ultrasound at 40,000 Hz, and the ultrasound signal is received in the echo pin. A test may override the echo signal of the sensor to create a simulated situation in which the robot detects a wall or an obstacle. The trigger pin is programmed to send an ultrasound continuously in this robot, which is independent of the simulation, so we use a single channel to emulate the echo signal.

To fully automate the testing process, we removed the chassis part from the Arduino-based robot, which does not influence the states of the PWM channels but prevents the robot from moving physically. Because our test oracles are based on the PWM signals of the DC motors to examine the robot's behaviour without the information of the physical location. For example, we designed the assertion for the robot turning left as *right_fwd_pwm* > *right_fwd_pwm*, where the forward PWM signal of the right motor is greater than that of the left motor. As a consequence, the whole mutation testing process is automated and requires no human observations.

RESULT

The overall mutation scores of the four implementations are quite similar, i.e., 0.34, 0.36, 0.39 and 0.40. The test suites examine the five movements of the robots; they are almost the same for the four student projects that we consider. Table 5.8 summarises the mutants for these four implementations. We observe that 416 mutants have been generated. We did not find equivalent mutants amongst the generated mutants (**RQ5.2**). This

Table 5.8: Mutants result of line-follower robot

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	38	38	26	12	0	0.32
OSR	34	34	25	9	0	0.26
PNR	298	298	184	114	0	0.38
IVR	0	0	0	0	0	-
EDR	0	0	0	0	0	-
IOMR	36	36	19	17	0	0.47
SIR	4	4	2	2	0	0.50
SOR	3	3	2	1	0	0.33
SVR	3	3	3	0	0	0.00
Overall	416	416	261	155	0	0.37

Note: Column "MOP" means mutation operator. Column "#Equiv." means the number of the equivalent mutants. Column "MS" means nutation score.

is likely due to the control program of the Arduino being quite simple: it is mainly a signal receiver for the ROS core. The key program, the image processing program, on the other hand, is located on the PC side.

For **RQ5.1** we note that while the statement coverage of the test suite is 100%, the overall mutation score is 0.37. Further investigation of the test suite leads us to the fact that the existing test suite lacks assertions to examine all the target pins. In fact, the test suite only checks the states of two pins which control the forward direction of the motors (i.e. the 1FWD and 2FWD ports in the LCHB-100 H-bridge). Ideally, the test suite should check the four pins connected to the other ports of the LCHB-100 H-bridge.

To kill the alive mutants (**RQ5.3**), we added four assertions in each test case to ensure the correct states of the pins connected to the LCHB-100 H-bridge that controls the movement of the motors. This improvement resulted in 201 mutants being killed. However, there are still 60 mutants surviving after the modification. These 60 mutants are hard to kill due to the limitations of our test environment setup. Among the 60 stubborn mutants, 20 mutants are related to a pin that the hardware monitor did not track. This pin is to control an LED which students mostly used for debugging purposes. These 20 mutants could be killed if we monitor the states of the LED pin and add specific assertions for it. The remaining 40 mutants are hard to kill because our test environment can only monitor the pin states of the robot. This means that we cannot further check the other settings of the pins, e.g., the pin mode and resistor state, as we can do in the Raspberry Pi platform. This type of stubborn mutants is different from the previously observed stubborn mutants in the *hcsr04sensor* and *four-wheel robot* projects, where the stubbornness was due to software testability issues. As mentioned in Section 5.5.2, limitations of the Arduino platform prevent us from touching the codebase of the control program directly. The adoption of the hardware monitor treats the system as a black box; this restricts the features that we can test in this system, such as the internal settings of the pins. For this line-following robot, 90.4% non-equivalent surviving mutants can be killed by adding extra test cases, while the rest mutants are not killable due to test setups.

Table 5.9: Mutants result of all subjects

MOP	#Generated	#Covered	#Alive	#Killed	#Equiv.	MS
OVR	90	85	52	38	0	0.42
OSR	83	80	51	32	1	0.39
PNR	685	673	264	421	0	0.61
IVR	29	28	4	25	0	0.86
EDR	19	19	0	19	0	1.00
IOMR	73	72	21	52	0	0.71
SIR	15	15	5	10	0	0.67
SOR	18	18	9	9	0	0.50
SVR	24	24	19	5	11	0.38
Overall	1036	1014	425	611	12	0.60

Note: Column “MOP" means mutation operator. Column “#Equiv." means the number of the equivalent mutants. Column “MS" means nutation score.

5.5.3. SUMMARY

Based on the case studies on the Raspberry Pi and Arduino platforms, we evaluated our method in terms of the efficiency in generating non-equivalent mutants (**RQ5.2**) and the effectiveness in evaluating the test suite quality (**RQ5.1**). Moreover, we also manually analysed non-equivalent surviving mutants to explore whether the mutation score can be improved by implementing new or improved tests (**RQ5.3**). In this section, we summarise all results of all subjects involved in our experimental study (as shown in Table 5.9) and answer the three research questions in the light of our observations.

Table 5.9 indicates that there are 1036 mutants generated in total, with the PNR mutants comprising 66.1% of the total. The EDR mutants are easiest to kill, while the OSR and SVR mutants are most difficult to kill. For **RQ5.2**, the overall percentage of non-equivalent mutants is 98.8%, which is quite promising. The equivalent mutants mainly stem from SVR (one from project *gpiozero* and ten from project *four-wheel robot*). However, the equivalent versions without the initial value setup are not recommended since different embedded platforms have different default values. Explicitly setting the initial value in the setup function can avoid unexpected initial states. The other equivalent one arises from OSR, which is due to dead code (see project *RPLCD* in Section 5.5.1). Besides, three SIR mutants are non-killable which are caused by the circuit of the peripherals. We considered these mutants as equivalent mutants in the context of physical computing systems. Even taking the three SIR mutants into consideration, the non-equivalent mutants still comprise 97.5% of the total number of mutants, showing MUTPHY has high efficiency in generating non-equivalent mutants.

For **RQ5.1**, compared to the statement coverage, the mutation score generated by our method can be a better indicator of test suite quality. More specifically, the mutation score can evaluate how well the test suite examines the integration part of the software and peripherals in physical computing systems, something the statement coverage does not allow. Except for project *gpiozero*, all the non-equivalent alive mutants reveal the inadequate test cases in the existing test suite. This is especially true for project *RPLCD*, for which the mutation score is 0, while the statement coverage is 71%.

For **RQ5.3**, 94.2% of the mutants, in most cases, it is possible to kill all non-equivalent surviving mutants by adding extra test cases, which again supports **RQ5.1** that mutation score can effectively evaluate the existing test suite. The exception being 59 mutants. The Raspberry Pi case studies account for 19 of these mutants: 2 mutants from project *hcsr04sensor* and 17 mutants from project *four-wheel robot*. Killing these mutants would require refactoring the production code to increase the observability of state changes. This implies that test quality is not the only factor to determine the mutation score, as the testability of the production code can also impact the mutation score. Moreover, introducing mock objects is a double-edged sword. If the mock objects are used properly, the behaviour of the GPIO pins cannot be examined, e.g., replacing the whole *RPi.GPIO* module to mock objects in project *RPLCD*. While proper use of mock objects can improve the observability of intermediate state changes to derive high-quality tests (see project *hcsr04sensor* and project *four-wheel robot*). For Arduino, 40 mutants remain not-killed as our test setups are unable to assess the internal settings of the system. A deeper analysis of these 40 mutants reveals that factors such as the testability of the software under test and the test setup influence the mutation score. We would like to explore these potential factors in the future work to further understand mutation testing and thus improve it.

5.6. THREATS TO VALIDITY

External validity: First, our results are based on the Raspberry Pi and Arduino platforms; these results might be different when using other embedded platforms. Second, concerning the subject selection, we only chose nine physical computing systems in total to evaluate our approach. Unfortunately, few physical computing systems on the Raspberry Pi and Arduino platforms with up-to-date test suites are publicly available.

Internal validity: The main threat to internal validity for our study is the implementation of MUTPHY for the experiment. To reduce internal threats to a large extent, we carefully reviewed and tested all code for our study to eliminate potential faults in our implementation. Another threat to internal validity is the detection of equivalent mutants through manual analysis. However, this threat is unavoidable and shared by other studies that attempt to detect equivalent mutants.

Construct validity: The main threat to construct validity is the measurement we used to evaluate our methods. We used the percentage of non-equivalent mutants and the mutation score as key metrics in our experiment, both of which have been widely used in other studies on mutation testing.

5.7. RELATED WORK

There has been a great deal of work on verification and validation of embedded systems (not limited to physical computing systems) in literature. The main methodologies are static analysis (e.g., [64, 357]), dynamic analysis (e.g., [309, 352]), formal verification (e.g., [209, 223]), black-box testing (e.g., [334, 341]), and white-box testing (e.g., [233, 383]).

Most related to our approach are *software-implemented fault injection (SWIFI)* techniques that inject faults pre-runtime at machine code level (e.g., by changing the content of memory/registers based on specified fault models) to emulate the consequences of hardware faults [152]. One of the earliest SWIFI techniques was presented by Segall et al. [316]. Their technique's initial results showed usefulness in reducing the fault injection complexity and validation of the system. Later, in 1995, Kanawati et al. [205] proposed a flexible software-based fault and error injection system, which is useful in evaluating the dependability properties of complex systems. More recently, Arlat et al. [58] compared physical and software-implemented fault injection techniques. As shown in their results, these two types of fault injection techniques are rather complementary, while SWIFI approaches are preferable mainly due to high controllability, repeatability and cost-effectiveness. All the above works focus on hardware testing, and more specifically, the kernel layer. None of them considers the communication between the software and peripherals in physical computing systems.

Concerning the application of mutation testing in embedded systems, Zhan et al. [376], He et al. [175] and Stephan et al. [330] have addressed the notion of *Simulink model mutations*. They proposed a set of mutation operators explicitly for Simulink that target the run-time properties of the model, such as signal addition operators. Moreover, Enoiu et al. [133] investigated mutation-based test generation for PLC embedded software using model checking. In their work, they designed six mutation operators for PLC embedded software relying on commonly occurring faults in IEC 61131-3 software [280, 321]. Different from our approach, all these works target mutation testing at the model level, and can only be applied to one specific type of software, e.g. *Simulink*. Our approach, on the other hand, is based on source code, and can thus potentially apply to all kinds of embedded system platforms.

5.8. Conclusion & Future Work

Physical computing systems come with their own set of challenges. This chapter focuses on the challenge of testing these physical computing systems, with a particular focus on assessing the quality of the tests that validate the interactions between the software and the physical components. We zoom in on common mistakes that occur in these interactions and propose a novel mutation testing approach with nine mutation operators targeting these common interaction mistakes.

Our results have shown encouraging results in uncovering weaknesses in existing tests. As such, our mutation testing approach enables to guide engineers to test systems more effectively and efficiently. More specifically, for our nine case study systems our mutation testing tool generated a total of 1036 mutants of which 41% were not killed by the original test suite (and 1.2% of the overall mutants being equivalent mutants). Adding tests or reinforcing existing tests made it possible to kill 94% of the non-equivalent surviving mutants.

This chapter makes the following contributions:

- a generic mutation testing approach for physical computing systems;
- a mutation testing tool named MutPhy working on the Raspberry Pi and Arduino platforms;
- a preliminary experiment on nine physical computing systems[2];

Future work. In the future, we aim to conduct additional case studies on more realistic physical computing systems. Also, we would like to explore the complementarity between traditional mutation operators and our newly designed, yet very specific mutation operators. Finally, we also aim to explore the relationship between testability and mutation score.

[2]All the tools, scripts and metadata for this experimental study are available in our GitHub repository [388].

6
Massively Parallel, Highly Efficient, but What About the Test Suite Quality? Applying Mutation Testing to GPU Programs

Thanks to rapid advances in programmability and performance, GPUs have been widely applied in High-Performance Computing (HPC) and safety-critical domains. As such, quality assurance *of GPU applications has gained increasing attention. This brings us to mutation testing, a fault-based testing technique that assesses the test suite quality by systematically introducing small artificial faults. It has been shown to perform well in exposing faults. In this chapter, we investigate whether GPU programming can benefit from mutation testing. In addition to conventional mutation operators, we propose nine GPU-specific mutation operators based on the core syntax differences between CPU and GPU programming. We conduct a preliminary study on six CUDA systems. The results show that mutation testing can effectively evaluate the test quality of GPU programs: conventional mutation operators can guide the engineers to write simple direct tests, while GPU-specific mutation operators can lead to more intricate test cases which are better at revealing GPU-specific weaknesses.*

This chapter has been published in the IEEE International Conference on Software Testing, Verification and Validation (ICST) 2020 [395].

6.1. Introduction

A graphics processing unit (GPU) is a single-chip processor originally used to boost the performance of video and graphics. The recent development of massive parallelism and energy efficiency and the ease of programming using the CUDA [267] and OpenCL [259] programming models have made GPUs attractive for High-Performance Computing (HPC), which requires compute-intensive, highly parallel computation [134, 336]. Moreover, GPUs are increasingly used in some safety-critical domains, such as medical science [332] and automotive [225]. In both HPC and safety-critical domains, *quality assurance* of GPU applications is an important issue [158, 303].

However, it is not easy to write a correct GPU program [226]. The essential elements of GPU programs are kernels, which are functions executed on GPU cores. To efficiently schedule instances of kernels on a GPU platform, a programmer needs to face the challenges in *computation* and *memory access* that do not appear in Central Processing Unit (CPU) programs.

Regarding computation, a programmer tends to spend less effort on thread management on the CPU platform than on the GPU platform [226]. To ensure high execution efficiency, a CPU usually supports far fewer threads running in parallel than a GPU of the same generation. Since the operating system needs to spend extra time on thread scheduling when the number of threads triggered by an application exceeds the total number of cores on the CPU. As a result, a CPU programmer may focus on the correctness and efficiency of a single thread, and leave the thread collaboration issues to libraries (e.g., OpenMP) and third-party tools. In contrast, managing threads on GPUs is challenging and rarely automated as a GPU often contains thousands of cores. Development toolkits as CUDA require programmers to manage the threads explicitly.

With regard to memory access, the hardware-level facilities for memory, such as the memory hierarchy, are typically transparent to programmers for the CPU. For instance, a CPU programmer may reorder entries in an array without knowing or controlling the movement of data in between the main memory and CPU cache. In contrast, a GPU platform has a separate memory space which is isolated from the main memory of the host computer. To ensure correctness and execution efficiency, GPU programmers have to explicitly manage different types of memory including shared memory, global memory and the memory of host computer [250].

Given the increasing demand for quality assurance of GPU applications as well as the challenges in GPU programming, it becomes essential to understand to which extent a GPU program can be analysed and tested. This brings us to *mutation testing*, a fault-based testing technique that measures the test effectiveness in terms of fault detection [197**?**]. Mutation testing has been shown to perform well in exposing faults compared to other test coverage criteria [144, 228, 244]. Also, mutants can act as a valid substitute to real faults [50, 201].

In this chapter, we aim to enable mutation testing for GPU programming to investigate if mutation testing can help in GPU program testing. To achieve this goal, we develop a mutation testing tool named MutGPU especially for GPU applications in the CUDA programming model. Considering the differences between CPU and GPU programming, we design nine new GPU-specific mutation operators in addition to conventional mutation operators. We perform an empirical study involving six GPU projects

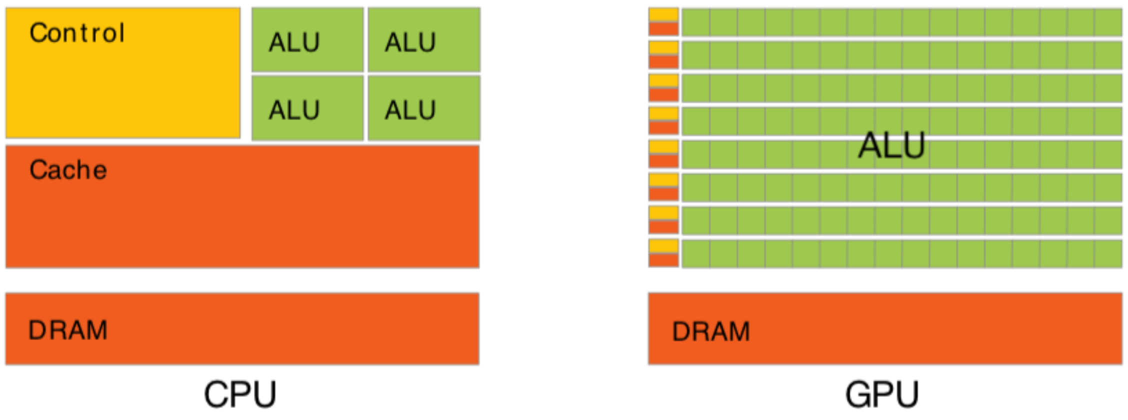


Figure 6.1: Comparison of CPU and GPU [267]

from the CUDA SDK [6]. To steer our experimental study, we propose the following research questions :

RQ6.1 *How frequently can GPU-specific mutation operators be applied, and how good is the existing test suite at killing them?*

RQ6.2 *How effective are conventional mutation operators in evaluating the test suite of GPU programs?*

RQ6.3 *How effective are GPU-specific mutation operators in evaluating the test suite of GPU programs?*

RQ6.4 *How do GPU-specific mutation operators compare with conventional mutation operators in terms of the improvement?*

6.2. BACKGROUND

6.2.1. GPU COMPUTING

A graphics processing unit (GPU) was originally dedicated to providing a high-performance, visually rich, interactive 3D experience [234]. With rapid advances of programmability and performance, a GPU becomes a compelling platform for computationally demanding tasks in various domains [283].

GPU computing, also known as general-purpose computing on the GPU (GPGPU), is to use a GPU as a co-processor to accelerate CPUs for general-purpose scientific and engineering computing [283]. Compared to the CPU, the GPU contains many more transistors devoted to data processing rather than data caching and flow control (as demonstrated in Figure 6.1) [267]. Thus, the GPU is especially well-suited for compute-intensive, highly parallel computation.

Another important characteristic of GPUs is that the programmable units (a GPU core) follow a “SPMD" programming model: single program, multiple data [283]. More specifically, the GPU processes many elements (vertices) in parallel using the same program, and each element is independent of the other elements.

```
void sum(int n, float *a, float *b, float *c){
  for (int i = 0; i < n; i++){
     c[i] = a[i] + b[i];}}
int main(){
  ...
  int N = 1<<20;
  // Perform SUM on 1M elements
  sum(N, a,b,c);
  ...}
```

Listing 6.1: *sum* function in Standard C

So far, there are three major GPU programming models: OpenCL (Open Computing Language), from the Khronos Group [259]; CUDA, from NVIDIA [267]; and C++ AMP, from Microsoft [164]. CUDA, as a popular GPU platform and programming model, was introduced by NVIDIA in November 2006. CUDA comes with a software environment that enables developers to program in C/C++. Except for C/C++, CUDA also supports Fortran, DirectCompute, OpenACC and Python.

6.2.2. EXAMPLE OF GPU PROGRAMMING

In this section, we are going to demonstrate the main concepts of GPU programming along with a simple program in *CUDA C*[1]. This program is to sum two vectors (*a* and *b*) into a third vector (*c*). In standard C, we can easily compute within a *for* loop shown as Listing 6.1. In CUDA C, we can accomplish the same addition on a GPU by introducing a *device* function. Here, we refer to the CPU and the system memory as the *host* and the GPU and its memory as the *device*. As shown in Listing 6.2, we add *__global__* to *sum()* (Line 1 in Listing 6.2) in order to notify the compiler that this function should be compiled to run on a device instead of the host. Moreover, we need to determine how many parallel copies of *sum()* function (named a *kernel* meaning a function that executes on the device) to launch. A kernel is executed by a collection of thread blocks (called a *grid*), and a thread block can be further split into threads (shown in Figure 6.2). The statement *sum<<<4096,256>>>* (Line 13 in Listing 6.2) specifies to launch 1,048,576 parallel threads (4096 blocks × 256 threads per block) for function *sum*.

After we launch the kernel with 1M parallel threads, the CUDA runtime assign varying values to those threads by *blockIdx.x**blockDim.x* + *threadIdx.x*[2] (Line 2 in Listing 6.2), the first taking *0* and the last taking *n-1*. Thus, all the threads run the same instructions but with different indices in parallel. Figure 6.3 presents the actual code being executed in threads.

Last but not least, in order to call the kernel, we first need to load the values of the two vectors (*a* and *b*) to the device (*dev_a* and *dev_b*) by invoking *cudaMemcpy* (Line 10 and 11 in Listing 6.2). The function *cudaMemcpy*, similar to *memcpy* in standard C, controls the memory copy between the device and the host. After the execution of the device function in the GPU, we then copy the output from the device (*dev_c*) to the host (*c*) (Line 14 in Listing 6.2).

[1]CUDA C is standard C with some ornamentation to allow developers to specify which code should run on the GPU and which should run on the CPU [311]

[2]Those are built-in variables in CUDA runtime that contains the value of thread index for whichever thread is currently running the device code.

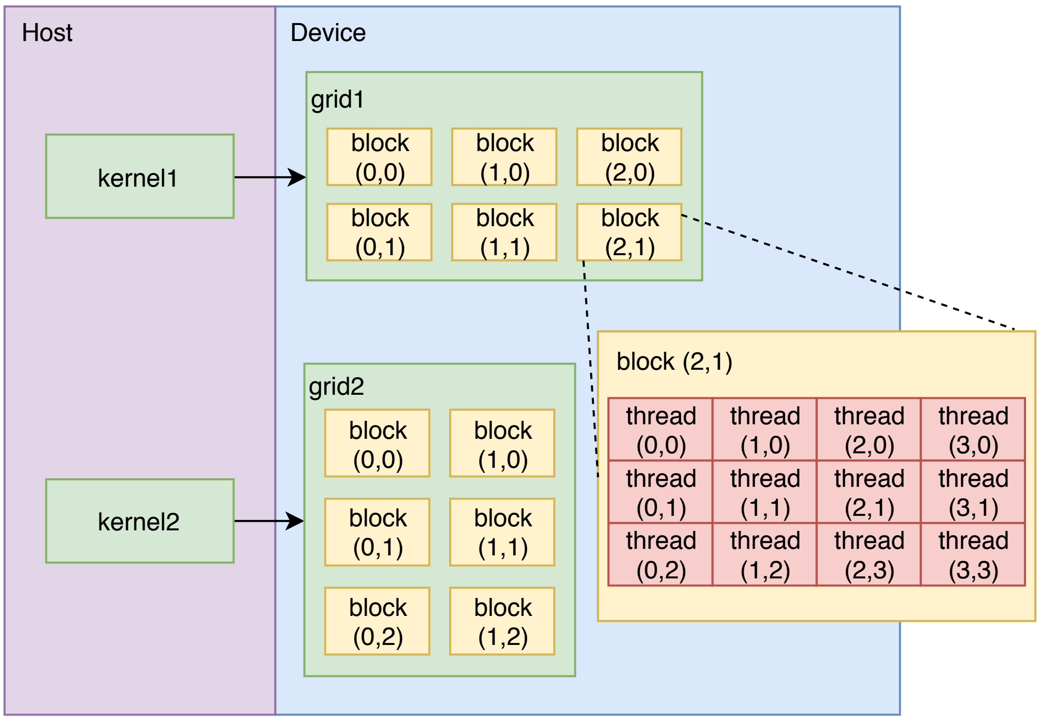


Figure 6.2: CUDA programming model [267]

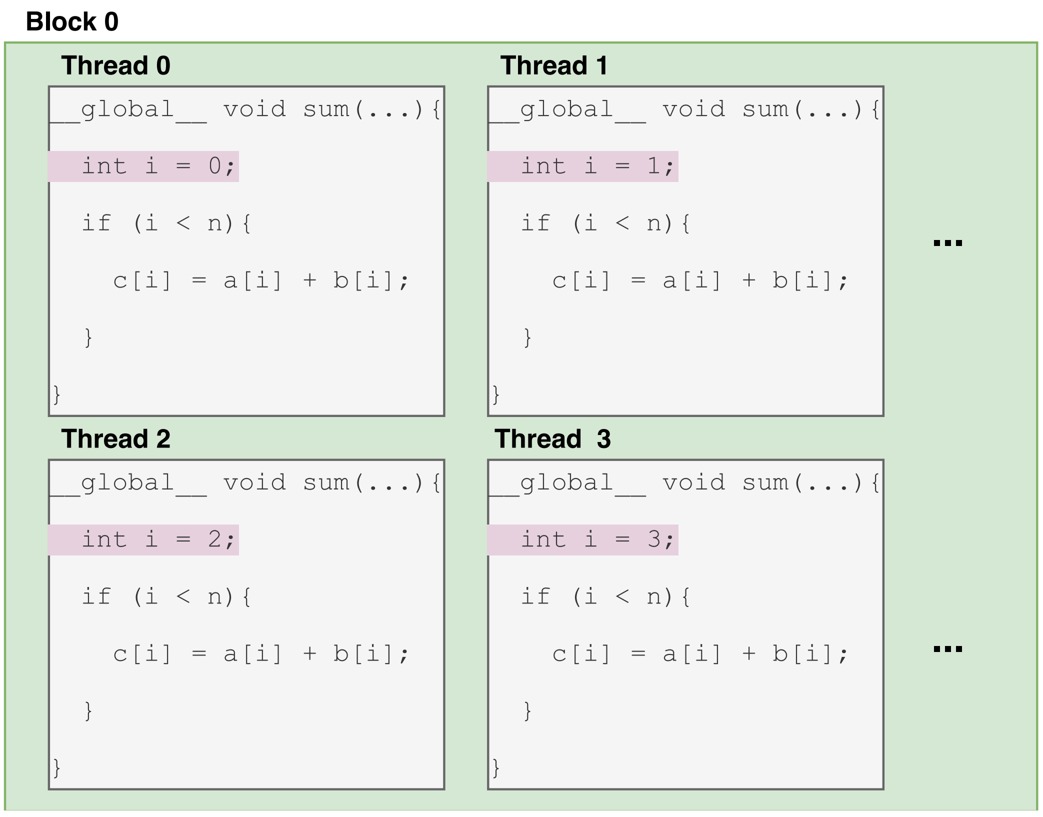
6.3. MOTIVATION

From Section 6.2, we can see there are important differences in the programming models of the CPU and the GPU. This raises the question of whether conventional mutation operators for C/C++ are enough to represent bugs in GPU programming? We observe that a test suite mutated with conventional operators may be insufficient: the GPU code with certain issues can still easily pass the test suite.

The first example is the *memory management* in GPU. In GPU programming, we first need to specify the number of parallel processors to launch in the device. In Listing 6.2, we can see that 1,048,576 parallel threads are used for the kernel. In this example, the size of our testing data is 2^{20}, the exact same size as the number of parallel threads; this means that every thread can process one index of the vectors. Thereby, this test can easily pass. What if the testing data exceeds 2^{20} (such as 2^{25} shown in Listing 6.3)? For this test case, the specific parallel threads are not enough to iterate and compute each index individually. Therefore, the test in Listing 6.3 fails. This exposes a bug in the *sum* function: we have to modify the CUDA C code to allow certain threads to compute more than one index of the vectors (see in Listing 6.4). For Listing 6.2, existing mutation operators for C cannot target the problem related to parallel processor allocation in GPUs.

Another instance we observe is the *thread management* in the GPU. Different from the CPU, GPU computing involves massive parallel operations via threads; the bugs in thread management are hard to represent with conventional mutation operators. For example, Listing 6.5 presents one example of indexing bugs that could occur in GPU programs.

We would also like to mention *atomic operations* in GPU programming. When devel-



```
__global__ void sum(...){
  int i = 0;
  if (i < n){
    c[i] = a[i] + b[i];
  }
}
```

```
__global__ void sum(...){
  int i = 1;
  if (i < n){
    c[i] = a[i] + b[i];
  }
}
```

```
__global__ void sum(...){
  int i = 2;
  if (i < n){
    c[i] = a[i] + b[i];
  }
}
```

```
__global__ void sum(...){
  int i = 3;
  if (i < n){
    c[i] = a[i] + b[i];
  }
}
```

Figure 6.3: Actual code in CUDA parallel threads

oping conventional single-threaded applications, there is no need for atomic operations. However, for GPU applications which are multithreaded by default, we do need a way to perform read-modify-write without being interrupted by another thread in certain conditions, such as reduction. Atomic operation omissions are one common mistake that happens in GPU programming, e.g., Listing 6.6.

To sum up, we do see a necessity to investigate mutation testing specifically for GPU programming.

6.4. Mutation Operators for GPU Programming

6.4.1. GPU-Specific Mutation Operators

Mutation operators are well-defined rules to specify the syntactic changes to generate faulty versions (called *mutants*) [279]. They are the key to mutation testing, where good mutation operators lead to effective test suites while poor mutation operators generate many trivial and redundant mutants.

To design mutation operators, we usually follow two methodologies [121]: the first is based on fault models, and the other is to analyse the syntax of the language being mutated. In our study, we mainly follow the later guideline: we have defined GPU-specific mutation operators based on the core syntax differences between CPU and GPU programming (as discussed in Section 6.3). Meanwhile, we also consider the syntactic changes in ways that programmers could make mistakes in GPU programming (the first

```
__global__ void sum(int n, float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n){
    c[i] = a[i] + b[i];}}
int main(){
  ...
  int N = 1<<20;
  int *a, *b, *c; // memory in host
  int *dev_a, *dev_b, *dev_c; // memory in device
  cudaMemcpy(dev_a, a, N, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, N, cudaMemcpyHostToDevice);
  // Perform SUM on 1M elements
  sum<<<4096,256>>>(N, dev_a, dev_b, dev_c);
  cudaMemcpy(c, dev_c, N, cudaMemcpyDeviceToHost);
  // examine the correct answer in c
  ...}
```

Listing 6.2: *sum* function in CUDA C

```
void test(){
  ...
  int N = 1<< 25 ;
  int *a, *b, *c; // memory in host
  int *dev_a, *dev_b, *dev_c; // memory in device
  cudaMemcpy(dev_a, a, N, cudaMemcpyHostToDevice);
  cudaMemcpy(dev_b, b, N, cudaMemcpyHostToDevice);
  sum<<<4096,256>>>(N, dev_a, dev_b, dev_c);
  cudaMemcpy(c, dev_c, N, cudaMemcpyDeviceToHost);
  ...}
```

Listing 6.3: A test case for *sum* function (Listing 6.2)
Note: we highlight the different test data setting in yellow colour.

guideline). To verify that the mutation operators we have proposed represent the common mistakes in GPU programming, we have searched for them on *StackOverflow* with the keyword “cuda" + issue name. For example, the keyword for the shared memory issue is “cuda shared memory". After searching for “cuda shared memory”, we have obtained 500 search results from StackOverflow sorted by relevance. We have analysed the first 150 items. Among those, 48% are referring to bug issues.

We categorise the GPU-specific mutation operators according to the key syntactic differences between CPU and GPU programming. In the following sections, we describe the GPU-specific mutation operators we proposed by category.

MEMORY MANAGEMENT

Execution configuration As mentioned in Section 6.2.2, we need to specify the *execution configuration* for a kernel function. The execution configuration defines the dimension of the grid and blocks that will be used to execute the function on the device. To de-

```
__global__ void sum(int n, float *a, float *b, float *c){
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  while (i < n) {
    c[i] = a[i] + b[i];
     i += blockDim.x * gridDim.x; }}
```

Listing 6.4: Modified *sum* function in CUDA C
Note: we highlight the modified parts in yellow colour.

```
__global__ void sum(int n, float *a, float *b, float *c){
  int i = threadIdx.x*blockDim.x + blockIdx.x ;
  while (i < n)}{
    c[i] = a[i] + b[i];
    i += blockDim.x * gridDim.x; }}
```

Listing 6.5: *sum* function in CUDA C with indexing bugs
Note: we highlight the faulty parts in yellow colour.

```
__global__ void histogram(unsigned char *buffer,
            long size, unsigned int *histo){
 int i = blockIdx.x*blockDim.x + threadIdx.x;
 while (i < n){
  histo[buffer[i]] += 1; i += blockDim.x * gridDim.x;}}
```

Listing 6.6: *histogram* in CUDA C with atomic operation omission
Note: we highlight the faulty parts in yellow colour.

termine the number of parallel processors allocated for the kernel, many factors should be taken into consideration, for instance, the maximum size of the data and the limitation of the GPU device. For the execution configuration is unique in GPU programming, we propose the following three mutation operators to cover this aspect:

- *alloc_swap*: to replace the number of threads with the number of blocks in parallel processor allocations (and vice versa). The same bug was posted in *SO29158775* [33].

```
add<<<4096,256>>>(N, a, b, c); // original
→ add<<<256,4096>>>(N, a, b, c); // mutant
```

Listing 6.7: Example of *alloc_swap* mutator

- *alloc_increment*: to increase the number of parallel processors (in both threads and blocks) allocated by one

```
add<<<4096,256>>>(N, a, b, c); // original
→ add<<<4096+1,256>>>(N, a, b, c); // mutant
```

Listing 6.8: Example of *alloc_increment* mutator

- *alloc_decrement*: to decrease the number of parallel processors (in both threads and blocks) allocated by one

```
add<<<4096,256>>>(N, a, b, c); // original
→ add<<<4096-1,256>>>(N, a, b, c); // mutant
```

Listing 6.9: Example of *alloc_decrement* mutator

Shared memory The shared memory in GPU programming provides a means by which threads within a block can communicate and collaborate on computations [267]. To declare the variable in shared memory, we use the *__shared__* memory space specifier in CUDA C, for instance, *__shared__ float share[64];*. The shared memory management is the main cause of *data races* and *bank conflicts*. The mutation operator *share_removal* is introduced to represent such bugs in GPU programs. There are a great deal of questions posted on StackOverflow addressing the confusion about the shared memory in GPU programming, e.g., *SO25255699* [32] and *SO9488590* [36].

- *share_removal*: to remove the shared memory space specifier in variable declarations

```
  __shared__ float cache[N];  // original
→ float cache[N];  // mutant
```

Listing 6.10: Example of *share_removal* mutator

THREAD MANAGEMENT

GPU indexing GPU programming introduces a new indexing mechanism to iterate the data and threads using built-in variables such as *threadIdx.x* and *blockIdx.x* (as mentioned in Section 6.3). While in conventional imperative programming, we use the loop statement (e.g., *for* and *while*) to iterate the data and threads. Since the indexing scheme is quite different from serial code, there are numerous questions posted on StackOverflow addressing the confusion about GPU indexing, such as *SO9859456* [37], *SO21677559* [31] and *SO33159171* [35]. Therefore, we design three mutation operators address the indexing issue:

- *gpu_index_replacement*: to replace the thread indexing variable (*threadIdx*) with the block indexing variable (*blockIdx*) and vice versa

```
  int tid = blockIdx.x;  // original
→ int tid = threadIdx.x;  // mutant
```

Listing 6.11: Example of *gpu_index_replacement* mutator

- *gpu_index_increment*: to increase the indexing variables (*threadIdx* and *blockIdx*) by one

```
  int tid = blockIdx.x;  // original
→ int tid =  blockIdx.x+1;  // mutant
```

Listing 6.12: Example of *gpu_index_increment* mutator

- *gpu_index_decrement*: to decrease the indexing variables (*threadIdx* and *blockIdx*) by one

```
  int tid = blockIdx.x;  // original
→ int tid =  blockIdx.x-1;  // mutant
```

Listing 6.13: Example of *gpu_index_decrement* mutator

Synchronisation functions The synchronisation function, also called *barrier*, is used to coordinate communications between threads in a specific block (e.g., *__syncthreads()* function in CUDA C). Many *data races* and *deadlocks* are caused by the incorrect barrier placement. We propose *sync_removal* to mimic the mistakes of the incorrect barrier placement. The omission or misplacement of the synchronisation function are very common in GPU code, e.g., *SO29233426* [34].

- *sync_removal*: to remove the synchronisation function call (e.g. *__syncthreads()*)

```
int i = blockDim.x/2;
while (i != 0) {
  if (cIndex < i)
    cache[cIndex] += cache[cIndex + i];
  __syncthreads();  //original
→ //__syncthreads();  //mutant
  i /= 2;}
```

Listing 6.14: Example of *sync_removal* mutator

ATOMIC OPERATIONS

Atomic operations are unavoidable in multithreaded applications in the sense that they are guaranteed to be performed without interference from other threads. In other words, no other thread can access this address until the operation is complete. However, if the programmer does not pay enough attention, he or she is very likely to omit the atomic operations in GPU programming, for example, *SO14057678* [30].

- *atom_removal*: to remove the atomic functions (e.g., *atomicAdd()*) with non-atomic operations

```
while (i < size) {
 atomicAdd(&(histo[buffer[i]]),1); //original
→ histo[buffer[i]]) += 1; //mutant
  i += stride;}
```

Listing 6.15: Example of *atom_removal* mutator

6.4.2. CONVENTIONAL MUTATION OPERATORS

CUDA C is an extension of standard C [267], therefore, conventional mutation operators for C also apply to CUDA C. Since the existing C mutation tools, such as Mull [122], cannot fully parse the grammar of CUDA C, we have to define the grammar of CUDA C first. So far, we have implemented five mutation operators which are most widely adopted in mutation testing, i.e., *conditional boundary replacement, negate conditional replacement, math replacement, increment replacement* and *logical replacement*.

6.4.3. GPU-SPECIFIC V.S. CONVENTIONAL MUTATION OPERATORS

The design principle of CUDA C/C++ is based on the traditional C/C++ syntax, which makes it easy to learn and use for developers. Although our newly proposed GPU-specific mutation operators seem to be subsumed by the existing mutation operators in terms of syntax, they are semantically different. Take *gpu_index_replacement* from GPU-specific mutation operators and *array_reference* from conventional mutation operators for example. The operator *gpu_index_replacement* replaces the thread indexing variable (*threadIdx*) with the block indexing variable (*blockIdx*). The thread/block indexing variable on the GPU is not equivalent as the array variable on the CPU. The thread/block indexing variable is used to access the parallel processors on the GPU, while *array_reference* variable is used to access the memory blocks on the CPU. One thing to notice here is that the mutants generated by GPU-specific mutation operators are totally different from the

Table 6.1: Summary of mutation operators

category	operator	definition
C	conditional_boundary_replacement	replace the relational operators <, ≤, >, ≥ with their boundary counterpart (according to PIT [103])
	increment_replacement	replace increments with decrements and vice versa
	logical_replacement	replace logical operator AND (&&) with OR (||) and vice versa.
	math_replacement	replace binary arithmetic operations with another operation (according to PIT [103])
	negate_conditional_replacement	replace the relational operators with another operation (according to PIT [103])
GPU	alloc_decrement	decrease the number of parallel processors (in both threads and blocks) allocated by one
	alloc_increment	increase the number of parallel processors (in both threads and blocks) allocated by one
	alloc_swap	replace the number of threads with the number of blocks in parallel processor allocations (and vice versa)
	atom_removal	remove the atomic functions (e.g. *atomicAdd()*) with non-atomic operations
	gpu_index_decrement	decrease the indexing variables (*threadIdx* and *blockIdx*) by one
	gpu_index_increment	increase the indexing variables (*threadIdx* and *blockIdx*) by one
	gpu_index_replacement	replace the thread indexing variable (*threadIdx*) with the block indexing one (*blockIdx*) and vice versa
	share_removal	remove the shared memory space specifier in variable declarations
	sync_removal	remove the synchronisation function call (e.g. *__syncthreads()*)

conventional operators; this means there is no overlap between those two sets of mutants.

To sum up, we have designed nine mutation operators (summarised in Table 6.1) to replicate common errors in GPU programming. Meanwhile, we have also implemented five conventional mutation operators that can be applied to CUDA C programs (also included in Table 6.1).

6.5. TOOL IMPLEMENTATION

As mentioned earlier in Section 6.2, there are three major GPU programming models, i.e., OpenCL, CUDA and C++ AMP. In this chapter, we select CUDA as the target model to implement the aforementioned mutation operators.

To evaluate our approach, we have implemented a prototype tool (coined MUTGPU) in Python to apply mutation testing in GPU programs. Figure 6.4 presents an overview of the architecture of MUTGPU [389]. MUTGPU consists of two components, i.e., the mutation engine and the test executor. MUTGPU takes the program and its test suite as input. First, the mutation engine analyses the source code and marks all possible mutation points, and then the mutation generator produces all the mutants according to mutation operators. After that, the program and generated mutants together with the test suite go to the test executor, where the mutation testing is performed: each mutant is executed against the test suite one by one. Finally, MUTGPU prints out the detailed mutant killable results.

The main task of the code analyser is to analyse the test dependencies and parse the source code of the program for the mutation generator. We have adopted *Pyparsing* [248] as the code analyser to parse the CUDA C code. *Pyparsing* is a pure-Python class library that constructs recursive-descent parsers with ease. The mutation generator contains all the mutation operators and the details of the mutants including the mutation location (line number) and the mutation operator type.

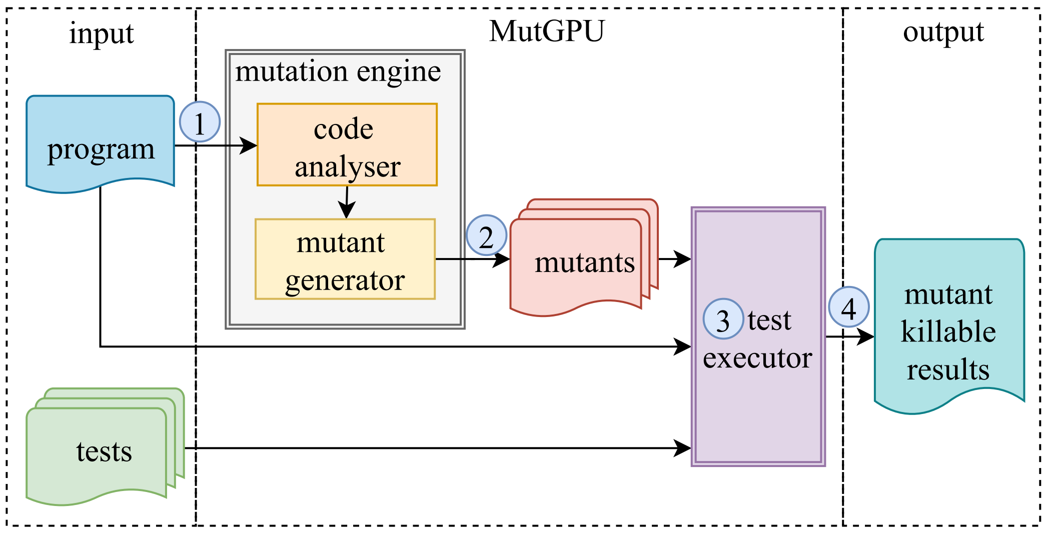


Figure 6.4: Overview of MutGPU architecture and workflow

6.6. Empirical Evaluation

To assess the efficacy of our mutation testing approach, we conducted an experimental study based on the CUDA programming model. The main purpose of this study is to investigate whether GPU programming can benefit from mutation testing, so we proposed the following research questions to steer our experimental study:

- **RQ6.1:** *How frequently can GPU-specific mutation operators be applied, and how good is the existing test suite at killing them?*
- **RQ6.2:** *How effective are conventional mutation operators in evaluating the test suite of GPU programs?*
- **RQ6.3:** *How effective are GPU-specific mutation operators in evaluating the test suite of GPU programs?*
- **RQ6.4:** *How do GPU-specific mutation operators compare with conventional mutation operators in terms of the improvement?*

6.6.1. Subject Systems

We select six GPU benchmark projects from the CUDA SDK [6]. All these benchmark projects are widely used in the research domain [78, 82, 227]. Table 6.2 summarises the main characteristics of the selected projects. All systems are written in CUDA C, and contain a set of test cases.

We perform the experiment on two different NVIDIA graphic cards (GeForce MX150 & GTX 960) with two releases of CUDA toolkit (9.0 & 9.1) to minimise the threat caused by errors residing in hardware and CUDA toolkits.

6.6.2. Experimental Setup

To answer **RQ6.1**, we investigate the mutant results for each GPU-specific mutation operator we proposed in detail. More specifically, we evaluate the *frequency* of each muta-

Table 6.2: Subject systems

Project	File	LOC		hCOV	#Mutants	
		Source	Test		C	GPU
MonteCarloMultiGPU	MonteCarlo_kernel.cu	231	359	100	71	59
	MonteCarlo_reduction.cuh	71			14	2
convolutionFFT2D	convolutionFFT2D.cu	226	509	100	40	36
	convolutionFFT2D.cuh	463			250	64
histogram	histogram64.cu	219	141	100	82	96
	Histogram256.cu	165			48	81
mergeSort	mergeSort.cu	636	95	100	264	300
transpose	transpose.cu	349	174	100	180	319
scan	scan.cu	290	116	100	90	70
	total	2650	1394	100	1039	1027

Note: Column "LOC" standing for the line of code is measured by sloccount[363]. Column "hCOV" indicates the statement coverage for the host code. For device code, coverage analysis is a wrong approach as already discussed by the Nvidia community [4].

tion operator based on the number of the generated mutants and the mutation score.

For **RQ6.2** and **RQ6.3**, we determine the effectiveness of the mutation operators in assessing test quality of GPU programs based on non-equivalent surviving mutants. Because non-equivalent surviving mutants are crucial to calculate the mutation score, and by investigating non-equivalent surviving mutants, we can see whether those mutants are due to inadequate test suites or not. We have implemented five conventional and nine GPU-specific mutation operators in our tool MutGPU. We apply all the mutation operators to the six subject systems and manually analyse the non-equivalent surviving mutants. We identify the equivalent mutants by hand.

To compare the conventional mutation operators with GPU-specific ones (**RQ6.4**), we are interested in what kind of enhancements the GPU-specific mutation operators can bring to the conventional mutation operators. In other words, we would like to investigate whether there exists some bugs or issues that cannot be detected by the conventional mutation operators, but can be detected by GPU-specific mutation operators. Therefore, we first try to manually engineer new test cases to kill all the possible conventional mutants to obtain a C-sufficient test suite for each system. Then, we apply GPU-specific mutation operators to the C-sufficient test suites to see if there are non-equivalent GPU mutants survived. The last step is to manually analyse the remaining GPU mutants that cannot be detected by the C-sufficient test suites.

6.7. Results

RQ6.1: Frequency of GPU-specific mutation operators & mutation scores

We sum up the mutant results for each mutation operator in Table 6.3. From Table 6.3, we can observe that operator *gpu_index_replacement* generates the most mutants (411) for GPU programs, followed by operator *gpu_index_decrement* (210) and *gpu_index_increment* (210). This indicates that the GPU indexing operations are commonly used in GPU programming. Thus, designing specific mutation operators for GPU indexing seems nec-

Table 6.3: Mutant results for each mutation operator

category	operator	#covered	#killed	#equiv.	#survived	#total	MS
C	conditional_boundary_replacement	114	60	45	58	118	0.822
	increment_replacement	12	9	0	3	12	0.75
	logical_replacement	5	2	3	3	5	1
	math_replacement	714	572	86	172	744	0.869
	negate_conditional_replacement	156	130	6	30	160	0.844
	subtotal	1001	773	140	266	1039	0.86
GPU	alloc_decrement	46	38	0	12	50	0.76
	alloc_increment	46	21	0	29	50	0.438
	alloc_swap	46	31	3	19	50	0.633
	atom_removal	1	1	0	0	1	1
	gpu_index_decrement	204	181	0	29	210	0.862
	gpu_index_increment	204	173	0	37	210	0.824
	gpu_index_replacement	403	340	0	71	411	0.827
	share_removal	20	17	0	3	20	0.85
	sync_removal	25	19	1	6	25	0.792
	subtotal	995	821	4	206	1027	0.803

Note: Column “#equiv." represents the number of the equivalent mutants. Column “MS" represents the mutation score which is calculated by the number of killed mutants divided by the number of non-equivalent mutants (the same in the following tables).

essary. The operator *atom_removal* only produces one mutant. We assume the reason behind the low mutant number (=1) is because the selected subject systems we selected do not contain many atomic operations.

From the aspect of the mutation score, except operator *atom_removal* whose mutation score is 1, the rest ranges from 0.438 to 0.862. This means that not all the mutants generated by the GPU-specific mutation operators can be detected by the existing test suites. There is still space for improvement in the existing test suites. The high mutation score for operator *atom_removal* is due to its low mutant number. This observation indicates that all GPU specific mutation operators we designed are useful in GPU programming to guide the engineers to write better tests.

The GPU-specific mutation operators we propose can be frequently applied in GPU programming. Furthermore, the mutation score obtained from applying these mutation operators ranges from 0.438 to 1.0.

RQ6.2: CONVENTIONAL MUTATION OPERATORS

Table 6.4 summarises the mutant results from running both C and GPU mutants against existing test suites. Overall, there are 1039 C mutants generated in total (as shown in Table 6.4). 96.3% mutants are covered by the existing test suites (meaning the line from which the mutant is generated is covered by the test suite), while 74.4% mutants (86.0% non-equivalent mutants) are killed. We can see that the mutation score (0.860) is lower than the mutation coverage (0.963), which means some mutants that are covered by the tests can still survive. Looking at the existing test suites, we found that most tests from the six projects do not target the unit-level: the main design of the test suites is to invoke a series of functions in turn and examine the final results in the end. Therefore,

Table 6.4: Mutant results

Project	File	C Mutants					GPU Mutants				
		#total	#covered	#killed	#equiv.	MS	#total	#covered	#killed	#equiv.	MS
MonteCarloMultiGPU	MonteCarlo_kernel.cu	71	65	44	7	0.688	59	59	23	0	0.390
	MonteCarlo_reduction.cuh	14	14	13	1	1.000	2	2	0	1	0.000
convolutionFFT2D	convolutionFFT2D.cu	40	40	40	0	1.000	36	36	27	0	0.750
	convolutionFFT2D.cuh	250	249	213	34	0.986	64	64	61	0	0.953
histogram	histogram64.cu	82	77	70	6	0.921	96	96	81	0	0.844
	Histogram256.cu	48	48	41	5	0.953	81	81	68	0	0.840
mergeSort	mergeSort.cu	264	244	180	33	0.779	300	288	265	0	0.883
transpose	transpose.cu	180	174	99	43	0.723	319	299	231	0	0.724
scan	scan.cu	90	90	73	11	0.924	70	70	65	3	0.970
	total	1039	1001	773	140	0.860	1027	995	821	4	0.803

Note: Column "#equiv." represents the number of the equivalent mutants. Column "MS" represents the mutation score which is calculated by the number of killed mutants divided by the number of non-equivalent mutants (the same in the following tables).

```
1 for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS){
2   if (xIndex < height && yIndex < width){
3     odata[index]=tile[threadIdx.y][threadIdx.x];}}
```

Listing 6.16: Bug example of an equivalent mutant (in *transpose.cu*)

it is very likely that a small change in the program (a mutant) do not propagate to the outputs (*fault masking* [162]). Moreover, *test directness*, which measures the extent to which the production code is tested directly, plays an important role in mutation testing (as mentioned in Chapter 4). However, there are *few* direct tests in the existing test suite to assess the difference between the original and the mutated programs; this causes a small number of mutants to not be detected by the existing test suites. The comparison of coverage and mutation score indicates that the mutation score is a stronger indicator of test suite quality than test coverage.

Speaking of equivalent mutants, the conventional mutation operators generate 140 equivalent mutants (13.5%) out of 1039 mutants as displayed in Table 6.4. Furthermore, from Table 6.3 where we sum up the mutant results for each mutation operator, we can see that over 50% of mutants (61.4%) are generated by operator *math_replacement*, followed by operator *conditional_boundary_replacement* (32.1%). Most of the equivalent mutants produced by *math_replacement* are because one operand in the math operation is zero, such as *threadIdx.x + 0*, thus the replacement of the math operators, e.g., from $+$ to $-$, does not influence the result. The equivalent mutants from *conditional_boundary_replacement* are owing to the fact that the boundary conditions (the equivalent condition, i.e., $=$) cannot be reached or satisfied. Also, there are a few equivalent mutants guiding us to detect the bugs in the systems. Listing 6.16 presents an example of a bug caught by the equivalent mutants. The mutant which replaces $<$ to $\leq$ in Line 1 is equivalent to the original program since the variable i does not affect the result of the loop. This equivalency turns out to be a potential bug in the program as variable i is useless in the loop. The cause for the bug is that there is a similar loop in the previous function (shown in Listing 6.17). Thus, the bug shown in Listing 6.16 might be due to copy-pasting the previous statements but omitting the modification. This finding supports that GPU programming can benefit from mutation testing.

```
for (int i=0; i<TILE_DIM; i+=BLOCK_ROWS){
  odata[index_out+i] = idata[index_in+i*width];}
```

Listing 6.17: Cause of bug example in Listing 6.16

Moreover, to further understand the effectiveness of the mutation operators in assessing test quality, we need to manually analyse the 126 surviving non-equivalent mutants to see whether these surviving non-equivalent mutants are due to inadequate test suites. Compared to conventional CPU programs, testing GPU code is more challenging. Only a small number of mutants (34 out of 126) can simply be killed by improving and adding tests. To detect the remaining mutants, we first need to refactor the existing code, and then add tests. The reason for a number of the mutants that need to be refactored is because many functions cannot be directly accessed from the test suites.

According to the CUDA programming model (shown in Figure 6.2), there are two parts in the GPU programs: the host (the CPU and the system memory) and the device (the GPU and its memory). The tests are mainly located in the host which invokes a function in the device/host and examines the result. Therefore, if the function is executed on the device and callable from the device only, it is impossible to access the function directly from the host. This exception is the function specified by *__device__*. The workaround is to wrap *__device__* functions with a *__global__* function which can be callable from the host. Other function specifiers, e.g., *static* and *inline*, also prevent the tests to access those functions from a different file. Since the access to *static* and *inline* functions is restricted to the file where they are declared. To test *static* and *inline* functions, we have to remove the *static* and *inline* specifiers to allow the access from a separate test file.

By modifying the function accessibility for the tests and adding direct tests, we can kill another 58 mutants. However, some mutants generated are located in intermediate variables which do not propagate to the output. These are much harder to detect: they usually require to split the method into smaller portions, or refactor the method to non-void. This might considerably alter the structure of the systems, and also affect the mutation score. Therefore, in this study, we leave this type of “stubborn" mutants [373] aside. Except for “stubborn" mutants, to kill a conventional mutant in a GPU program, adding one direct test without carefully choosing test input can work well which we can achieve it within 1 min. Finally, we can achieve 0.962 mutation score by improving the test quality, which indicates the conventional mutation operators can effectively evaluate the existing test quality in the context of GPU programming.

GPU programming can benefit from the conventional mutation operators which mainly guide the engineers to write direct tests for GPU programs. The average time to engineer a test case to kill a mutant is within 1 min.

RQ6.3: GPU-SPECIFIC MUTATION OPERATORS

From Table 6.4, we can see that there are 1027 mutants generated by GPU-specific mutation operators, slightly less than C mutants. Among all the GPU mutants, 96.9% mutants are covered by the existing test suites. The overall mutation score is 0.803. The same observation holds for GPU mutants as for CPU mutants: the mutation score is higher than

```
if (tid == 0){
    beta  = 0;beta2 = 0;
    for (int i = 0; i < blockDim.x; i += VEC) {
        beta  += sum[i];beta2 += sum2[i];}
    __TOptionValue t  = {beta , beta2};
    *d_CallValue = t;}
cg::sync(cta);
```

Listing 6.18: Bug example of an equivalent mutant (*MonteCarlo_reduction.cuh*)

the coverage. As we mentioned in Section 6.7, the majority of the tests only examine the final outputs after a series of function calls. Therefore, although a number of mutants are covered by the test suites, their changes do not propagate to the final outputs. This also shows the advantage of the mutation testing over test coverage in evaluating test quality.

In terms of equivalent mutants, GPU specific mutation operators only generate four equivalent mutants. Three are from operator *alloc_swap*, which are all located in the *execution configuration* for a kernel function call (*__global__* function call). The mutants generated by *alloc_swap* are due to the number of threads being the same as the number of blocks in parallel processor allocations. The last equivalent mutant is generated by operator *sync_removal*. This mutant is similar to Listing 6.16 which indicates the presence of a potential bug (presented in Listing 6.18). The main task of Listing 6.18 is to produce a smaller array of the sum result by *reduction*. The code fragment in Listing 6.18 does not contain write operations to the shared array *sum* and *sum2*, thereby, there is no need for a synchronisation function (*cg::sync(cta)*) in the end to guarantee that all of those writes to the shared arrays complete before anyone tries to read from the buffers. The synchronisation function (*cg::sync(cta)*) in Listing 6.18 is not necessary. This also confirms that misuses of the synchronisation functions in GPU programming are quite common, and the operator *sync_removal* can well represent common errors in GPU programming. Looking into the surviving GPU mutants, a large number of mutants (131) can easily be detected by modifying the function accessibility (e.g., remove *static* specifier) and adding direct tests as mentioned in Section 6.7. To kill the remaining non-equivalent mutants is more challenging: in addition to examining the expected result after the function call, more factors should be taken into consideration, such as the execution sequence, the size of the test input and the times of test execution. For instance, to detect the mutants from operator *sync_removal*, it usually requires multiple test executions (> 10 times), as the execution order of the code in GPU cores is undetermined. Although, we assume that GPU cores run the parallel program at the same time, there exist some latencies in different GPU cores. For instance, Thread 1 is executed before Thread 2 in one execution, while Thread 2 is executed before Thread 1 in another execution. Thus, *data races* do not occur every time. Another example is to kill the mutant generated from a *__global__* function. It is very likely to encounter illegal memory access if the test input size is inappropriate. To kill a GPU mutant, it usually requires us to understand the program context very well and choose more than one specific test input to kill a mutant; this could take up to hours to kill a GPU mutant. Thus, those tests designed to kill GPU mutants can better reveal GPU-specific weakness.

However, we found that not all the non-equivalent surviving mutants can be killed by adding test cases. There are 22 mutants not affecting the result of the kernel functions but the GPU performance. The GPU performance means the execution time by GPUs.

Table 6.5: Mutant results (C-sufficient test suites)

Project	File	C Mutants					GPU Mutants				
		t#otal	#covered	#killed	#equiv.	MS	#total	#covered	#killed	#equiv.	MS
MonteCarloMultiGPU	MonteCarlo_kernel.cu	71	71	61	7	0.953	59	59	49	0	0.831
	MonteCarlo_reduction.cuh	14	14	13	1	1.000	2	2	0	1	0.000
convolutionFFT2D	convolutionFFT2D.cu	40	40	40	0	1.000	36	36	27	0	0.750
	convolutionFFT2D.cuh	250	249	216	34	1.000	64	64	61	0	0.953
histogram	histogram64.cu	82	77	76	6	1.000	96	96	82	0	0.854
	Histogram256.cu	48	48	42	5	0.977	81	81	69	0	0.852
mergeSort	mergeSort.cu	264	264	213	33	0.922	300	300	280	0	0.933
transpose	transpose.cu	180	180	126	43	0.920	319	319	298	0	0.934
scan	scan.cu	90	90	78	11	0.987	70	70	65	3	0.970
	total	1039	1033	865	140	0.962	1027	1027	931	4	0.910

Note: Column “#equiv." represents the number of the equivalent mutants. Column “MS" represents the mutation score which is calculated by the number of killed mutants divided by the number of non-equivalent mutants (the same in the following tables).

For example, one mutant generated by *alloc_increment* modifies the number of threads in parallel processor allocations from 256 to (256-1). But the modifications do not influence the function output since the function already takes care of the boundary condition when the test input exceeds the number of parallel threads (just as in Listing 6.4). Also, the performance difference caused by the allocation decrement (-1) is too small to be sensed by any test. The performance difference is unique to GPU programming since the standard CPU programs do not use GPUs as co-processors. We suggest considering these mutants that only influence the performance at a small scale without output modification as equivalent mutants in the context of GPU programming. Another option to void such “equivalent" mutants generated from GPU programming could be using performance requirement to be part of the definition of a test case passing or failing.

GPU-specific mutation operators can effectively evaluate the test quality in the context of GPU programming. To kill the GPU mutants, many factors should be taken into consideration, such as test directness, the program context, the execution sequence and the test input size. It takes up to hours to kill a GPU mutant.

RQ6.4: CONVENTIONAL VS. GPU-SPECIFIC

In this section, we are going to shed light on the comparison of the conventional and GPU-specific mutation operators we proposed. As mentioned earlier, we are interested in what kind of improvements the GPU-specific mutation operators can bring to the conventional mutation operators. Thereby, we use C-sufficient test suites (100% mutation coverage for C mutants) as the base to apply mutation testing on the subject systems. Table 6.5 displays the mutant results based on the C-sufficient test suites. We can see that C-sufficient test suites have a high average mutation score using conventional mutation operators compared to GPU-specific ones; this is what we expect. Otherwise, GPU mutants are subsumed by C mutants, i.e., if the test suite achieve 100% mutation coverage on C mutants, it also achieves 100% mutation coverage on GPU mutants.

From Table 6.5, we can see that with an increase in mutation score with conventional mutation operators, the mutation score of GPU-specific mutation operators also

```
1  __global__ void padKernel_kernel(float *d_Dst, float *d_Src, int fftH, int
       fftW, int kernelH,int kernelW, int kernelY, int kernelX){
2    int y = blockDim.y * blockIdx.y + threadIdx.y;
3    int x = blockDim.x * blockIdx.x + threadIdx.x;
4    if (y < kernelH && x < kernelW){
5      int ky = y - kernelY;
6      if (ky < 0){
7        ky += fftH;}
8      int kx = x - kernelX;
9      if (kx < 0){
10       kx += fftW;}
11     d_Dst[ky*fftW+kx] = LOAD_FLOAT(y*kernelW+x);}}
```

Listing 6.19: Example of a surviving GPU mutant (*convolutionFFT2D.cuh*)

```
1  void test_padKernel_kernel(){
2    int N = 64*64;
3    float *d_Dst,*d_Src;
4    float h_Dst[N],h_Src[N],h_expected[N];
5    for(int i=0;i<N;i++){
6      h_Src[i]=i;
7      h_expected[i]=0;}
8    h_expected[3966]=1.0;
9    ...
10   cudaMalloc((void **)&d_Dst,sizeof(float)*N);
11   cudaMalloc((void **)&d_Src,sizeof(float)*N);
12   cudaMemcpy(d_Src, h_Src, N*sizeof(float), cudaMemcpyHostToDevice);
13   cudaMemset(d_Dst,0,N*sizeof(float));
14   dim3 threads(8,8);
15   → dim3 threads(32,8);
16   dim3 grid(iDivUp(3, threads.x), iDivUp(3, threads.y));
17   padKernel_kernel<<<grid,threads>>>(d_Dst,d_Src,64,64,1,1,1,1);
18   → padKernel_kernel<<<grid,threads>>>(d_Dst,d_Src,64,64,3,3,3,3);
19   cudaMemcpy(h_Dst, d_Dst, N*sizeof(float), cudaMemcpyDeviceToHost);
20   bool testFlag = true;
21   for(int i=0;i<N;i++){
22     if(h_expected[i]!=h_Dst[i]){
23       testFlag = false;}}
24       ...}
```

Listing 6.20: Direct test and its improved version for Listing 6.19
Note: we highlight the improved parts in yellow colour.

increases. This is mainly due to the design of the existing test suites, that do not target the unit testing level (mentioned in Section 6.7). We also observe a lack of direct tests for each function in the existing test suites. Thus, when we engineer new tests to kill the C mutants, we mainly concentrate on designing direct tests (as we discussed in Section 6.7). When we target a C mutant, the GPU mutant(s) located in the same line or the same function unit is also under investigation. Once the change(s) of the GPU mutant(s) in the same line or the same function unit can be observed by the direct test designed for the C mutant, the GPU mutant(s) can typically be killed at the same time.

Considering the remaining GPU mutants that are not detected by the C-sufficient test suites, the majority requires more delicate and complex test cases to observe the differences. Take a surviving GPU mutant in File *convolutionFFT2D.cuh* (shown in Listing 6.19) for example. The function *padKernel_kernel* aims to position the center of the convolution kernel at (0, 0) of the image which makes use of 2D GPU indexing.

All the mutants generated in Line 2 are detailed in Table 6.6. In order to kill the C mutant (MID = 0) in Line 2, we add a direct test (shown in Listing 6.20) for function *pad-*

Table 6.6: Mutant details for Listing 6.19 in Line 2

MID	operator	details	category	existing	c-sufficient
0	math_rep.	/	c	survived	killed
1	gpu_index_rep.	threadIdx.y	gpu	killed	killed
2	gpu_index_rep.	blockIdx.x	gpu	survived	survived
3	gpu_index_inc.	(blockIdx.y+1)	gpu	killed	killed
4	gpu_index_dec.	(blockIdx.y-1)	gpu	killed	killed
5	math_rep.	-	c	killed	killed
6	gpu_index_rep.	blockIdx.y	gpu	killed	killed
7	gpu_index_rep.	threadIdx.x	gpu	killed	killed
8	gpu_index_inc.	(threadIdx.y+1)	gpu	killed	killed
9	gpu_index_dec.	(threadIdx.y-1)	gpu	survived	survived

Kernel_kernel. However, the newly added direct test cannot detect the two GPU mutants with MID = 2 and MID = 9. Upon investigation, we found in the execution configuration for function *padKernel_kernel*, the numbers of blocks in the two dimensions are the same, i.e., *gridDim.x* = *gridDim.y* = 1 (see Line 14). Also, the condition in Line 4 of Listing 6.19 restricts the larger indexes of parallel threads for computation. Therefore, adding the direct tests cannot detect the difference of replacing *blockIdx.y* to *blockIdx.x*. To detect Mutant 2, we need to modify the numbers of blocks in the two dimensions to different values, e.g., set *gridDim.x* = 4 and *gridDim.y* = 1 (see Line 15). Moreover, we need to set the values of *kernelH* and *kernelW* (see Line 17&18) big enough so that the difference of the value *y* can affect the output of *d_Dst* array.

Together with this example, we can see that adding direct tests can kill most C mutants, but not all the GPU mutants. The remaining GPU mutants are more challenging to be killed as their differences with the original program are more subtle to tell. Only given test inputs with specific values and execution settings, these GPU mutants can be detected. This shows that the outputs of the GPU program are easily affected by test inputs and execution configurations. Thus, the corresponding test cases designed according to these types of GPU mutants are of higher quality and can detect more potential GPU-specific bugs in the systems. Therefore, it requires more effort to design delicate tests to kill a GPU mutants than the conventional C mutants.

Compared to conventional mutation operators, GPU-specific ones are better at guiding the engineers to design more delicate tests to detect the subtle differences in GPU programming. The efforts required to kill GPU mutants are higher than the conventional.

6.8. Threats To Validity

External validity: First, our results are based on the CUDA programming model; the results might be different when using other GPU programming models. Second, concerning the subject selection, we chose six GPU projects in total to evaluate our approach. All the projects are benchmark projects from the CUDA SDK [6], and are widely used in the research domain; this can minimise the threats caused by subjects. Moreover, there might exist errors in the GPU hardware and the CUDA toolkit. To avoid this threat, we conduct our experiment on different NVIDIA graphic cards and different releases of CUDA toolkit.

Internal validity: The main threat to internal validity for our study is the implementation of MutGPU for the experiment. To reduce internal threats to a large extent, we carefully reviewed and tested all code to eliminate potential faults in our implementation. Another threat to internal validity is the detection of equivalent mutants through manual analysis. However, this threat is unavoidable and shared by other studies that attempt to detect equivalent mutants [165, 252].

Construct validity: The main threat to construct validity is the measurement we used to evaluate our methods. We used the percentage of non-equivalent mutants and the mutation score as key metrics in our experiment, both of which have been widely used in other studies on mutation testing.

6.9. Related Work

With wide-spread applications of GPUs in High Performance Computing (HPC) [134, 336] and safety-critical domains (e.g., medical science [332] and automotive [225]), there have been increasing attentions on the *quality assurance* of GPU applications [158, 303], such as dynamic analysis [82] and formal verification [78, 227].

Most related to our approach are *fault injection* techniques in GPGPU applications. Farazmand et al. [137] attempted to quantify the Architectural Vulnerability Factor (AVF) of GPU hardware structures using statistical fault injection. They injected faults into register files, local memory, and active mask stack to characterise the vulnerability of different micro-architectural structures in GPUs to soft errors. Yim et al. [374] developed a mutation-based fault injection tool for automated reliability testing of GPU devices. They modeled both single- and multi-bit errors in the architecture state to represent the silent data corruption (SDC) error.

Fang et al. [135] proposed a fault injection methodology to evaluate the error resilience of the GPGPU applications. They aimed at injecting the faults that represent real *hardware* errors where they adopted the single-bit-flip fault model to simulate transient faults in GPU processors. Hari et al. [173] presented an fault injection-based framework called SASSIFI for GPU application resilience evaluation, especially on soft errors. SASSIFI serves two kinds of tasks: (1) inject bit-flip errors into the register file for AVF analysis; (2) inject errors in the outputs of the instructions for error propagation evaluation.

All the above studies have targeted the errors related to the GPU hardware. While we focus on the software aspect of GPU applications where we design mutation operators in the ways that engineers could make mistakes in GPU programming.

6.10. Conclusion and Future Work

This chapter aims to explore whether GPU programming can benefit from mutation testing. Compared to the CPU, the GPU differs greatly in the architecture and the programming model, thus, GPU programming comes with its own set of challenges. Upon observation, we found GPU code with issues of memory management, thread management and atomic operations can easily pass the test suite selected with conventional mutation operators. Thus, we propose nine GPU-specific mutation operators according to the main syntactic differences between CPU and GPU programming.

To evaluate our approach, we present a tool coined MutGPU and conduct an experiment on six CUDA systems. Our results show promising findings that GPU programming can benefit from mutation testing in three ways: (1) conventional mutation operators can guide engineers to write simple direct tests; (2) GPU-specific mutation operators can lead to more delicate test cases (thus higher quality and more test effort);
(3) equivalent mutants can help in bug detection.

This chapter makes the following contributions:

- nine GPU-specific mutation operators;
- a mutation tool (MutGPU [389]) working on CUDA;
- comparison of conventional and GPU-specific operators;
- a preliminary experiment on six GPU applications [389].

Future work. In the future, we aim to conduct additional case studies on more realistic GPU systems. Also, we would like to explore another GPU platforms, such as OpenCL.

7

Conclusion

This chapter concludes this thesis by first revisiting our research questions from Chapter 1, discussing the main threats to validity, and outlining the recommendations for future work.

7.1. Research Questions Revisited

In this section, we are going to revisit the research questions we propose in Chapter 1. We present the main takeaways from the answers to those research questions.

RQ2 *How is mutation testing actually applied?*

In Chapter 2, we have reported on a systematic literature review on the *application perspective* of mutation testing based on a collection of 191 papers from 22 venues. Through a detailed reading of this research body, we derive an attribute framework that is consequently used to characterise the selected studies in a structured manner. This attribute framework generalises and details the essential elements related to the actual application of mutation testing, such as in which circumstances mutation testing is used and which mutation testing tool is selected. In particular, a generic classification of mutation operators is constructed to study and compare the mutation operators used in the experiments described.

Our main findings from the systematic literature review are: (1) Most studies (70.2%) use mutation testing as as a fault-based evaluation method in different quality assurance processes. Those studies mainly target the unit level. (2) Many of the supporting techniques for making mutation testing truly applicable are still under-developed. The two key problems, namely the equivalent mutant detection problem and the high computation cost of mutation testing issues, are not well-solved in the context of our research body. (3) A deeper understanding of mutation testing is required, such as what particular kinds of faults mutation testing is good at finding. (4) The awareness of *appropriately* reporting mutation testing in testing experiments should be raised among the researchers. The first three findings further attract our attention, resulting in **RQ3 - RQ6**.

RQ3 *Can compression techniques be used to speed up mutation testing?*

In Chapter 3, we have conducted a detailed investigation of different compression techniques to speed up mutation testing using 20 open-source projects. We have introduced and investigated six compression strategies based on two clustering algorithms and three mutant selection strategies. In particular, we use *overlapped* and *FCA* groupings to cluster mutants based on their reachability and necessity condition against the test suite, and then apply mutant selection (three selection strategies, i.e., pure random, random with knowledge of mutation locations and random with mutation operator types) to decide which ones need to be executed in strong mutation. Also, we consider 10% random sampling and weak mutation methods as baselines.

The results of our empirical study show that mutant compression techniques can effectively speed up strong mutation testing up to 94.3 times with an accuracy > 90%. *FCA-based compression* is the fastest strategy, while *overlapped grouping* with the mutation location knowledge in mutant selection is the most accurate. In comparison, *weak mutation* attains a higher absolute error (23%) and lower accuracy (75%). *Random sampling* with 10% as sampling percentage is statistically less accurate than all mutant compression strategies, and worse in terms of speed-up than four compression strategies (excluding the two with knowledge of mutation locations).

Another important finding is that *mutation location trumps mutation operator information when selecting mutants to evaluate for strong mutation.* Hence, we suggest researchers should take into account the mutation locations in addition to the mutation operators when detecting redundant or subsuming mutants (e.g., [46, 202, 219]).

RQ4 *How can production code quality in terms of testability and observability influence the mutation score?*

In Chapter 4, we have investigated our hypothesis that code quality especially regarding the *testability* and *observability* plays an essential role in mutation testing. We have collected 64 existing source code quality metrics for testability, and have proposed a set of metrics that specifically target observability. We have first investigated the relationships between testability/observability metrics and the mutation score involving six open source Java projects. We observe that the 64 existing code quality metrics are not strongly correlated with the mutation score ($|rho|$ <0.27), while our newly proposed observability metrics do show a stronger correlation with the mutation score ($|rho|$ <0.5). In particular, *test directness*, which measures the extent to which the production code is tested directly, seems to be the most essential factor.

To better understand the causality of our insights, we have continued our investigation with a case study of 16 methods that scored particularly bad in terms of mutation score. In particular, we have refactored these methods and/or added tests according to the anti-patterns that we established in terms of the code observability metrics. We found that these anti-patterns can indeed offer software engineers actionable insights to improve both the production code and the test suite, and improve the mutation score along with it.

RQ5 *Can physical computing benefit from mutation testing?*

In Chapter 5, we have proposed a novel mutation testing approach for physical computing systems. We first have designed nine mutation operators based on *common mistakes* we have observed that are typically made by programmers during the implementation of software. To enable mutation testing for physical computing systems, we have implemented a mutation testing tool coined MUTPHY working on the Raspberry Pi and Arduino platforms. Then, we have conducted two case studies on both Raspberry Pi and Arduino platforms involving nine physical computing projects.

For nine systems, our mutation testing tool generates a total of 1036 mutants of which 41% are not killed by the original test suite (and 1.2% of the overall mutants being equivalent mutants). Adding tests or reinforcing existing tests makes it possible to kill 94% of the non-equivalent surviving mutants. Our results have shown encouraging results in uncovering weaknesses in existing tests. Thereby, we can see that physical computing could benefit from mutation testing in guiding engineers to test systems more effectively and efficiently.

RQ6 *Can mutation testing help in GPU program testing?*

In Chapter 6, we aim to enable mutation testing for GPU programming to investigate whether GPU programs can benefit from mutation testing. To achieve this goal, we have developed a mutation testing tool named MUTGPU especially for GPU applications in the CUDA programming model. Considering the differences between CPU and GPU programming, we have designed nine new GPU-specific mutation operators in addition to conventional mutation operators. We have performed an empirical study involving six GPU projects from the CUDA Code Samples [6]. Our results have shown that the conventional mutation operators can guide the engineers to write direct tests for GPU programs, while GPU-specific mutation operators can lead to more delicate and more complex test cases, which cannot be exposed by the conventional mutation operators.

7.2. THREATS TO VALIDITY

The threats to validity address the question of how our results and conclusions might be wrong [247], which is an essential part to judge the quality of a research study. In this section, we describe the main threats to validity of this thesis by summarising the threats to validity for each research question (**RQ2-RQ6**). There are detailed analyses of threats to validity for each research question in their corresponding chapter.

External validity: The *external validity* is concerned with whether we can *generalise* the results outside the scope of our thesis [139]. The main external validity for our thesis is the subject systems selection. The ways we have attempted to mitigate this threat to validity is (1) selecting at least six projects that differ in size, number of test cases and application domain; (2) selecting open-source projects whenever possible to enable replications of our study; (3) selecting projects that have already been widely used in related studies. Nevertheless, although we aim for generalisability, we do acknowledge that a broad replication of our study would mitigate any generalisability concerns even further.

Internal validity: The *internal validity* focus on how sure we can be that the treatment actually caused the outcome [139]. The main threat to internal validity for our thesis is in the self-implemented tools or scripts to conduct experimental studies. Except for the systematic literature review presented in Chapter 2, all the studies include at least a tool or script that we have implemented by ourselves. To eliminate potential faults in our implementations, we have carefully reviewed and tested all the code. Another threat to internal validity is the *equivalent mutant detection* [91] in our experiments. As a well-known *undecidable problem* [91], the *equivalent mutant detection* is unavoidable and shared by other studies on mutation testing that attempt to detect equivalent mutants or not [165, 252].

Construct validity: The *construct validity* concerns about the relation between the theory behind the experiment and the observation(s) [139], i.e., whether the the observed outcome corresponds to the cause/treatment we think or not. The main threat to construct validity is the measurement we have employed to evaluate our methods. Our attempts to minimise this risk is by adopting evaluation metrics that are widely used in research (such as recall, precision, and AUC), as well as a sound statistical analysis to assess the significance (Spearman's rank-order correlation).

7.3. Recommendations for future work

Although mutation testing has been investigated for over four decades, it is still facing numerous challenges for more broad applicability. This thesis aims to improve and extend the applicability of mutation testing, including speeding up mutation testing, deepening understanding of mutation testing and exploring new application domains. However, research on the applicability of mutation testing still needs a long way to go. In the following we envision four directions of future works based on the findings of this thesis.

- **R1: More tool support in integrated development environments (IDEs) is needed.**

  Most mutation testing tools, such as Mujava [240] and PiTest [13], are developed by third parties. The compatibility issue with the state-of-art IDEs (e.g., IntelliJ) is not well-solved. Developing seamless integration with IDEs (like has been done for JUnit) for mutation testing tools can greatly ease the difficulty of applying mutation testing in the daily work of developers, thus potentially accelerating the application of mutation testing.

- **R2: "Smart" and personalised mutation engine.**

  Most mutation testing tools provide the users with several options to choose from, for instance, the mutation operators to be used and which class/method to be mutated in the mutation testing. However, the mutation engine (i.e. the framework to create mutants) itself is not "smart" enough; this means the mutation engine just applies the selected mutation operators to the selected class/method mechanically to create mutants. Besides, every developer is unique and their common programming mistakes are various. It would be very useful that the mutation engine can learn from mistake patterns of each developer and create personalised mutants for him/her. Such a “smart" and personalised mutation engine not only speeds up the process of mutation testing, but also guides the developer to write better production code and tests.

- **R3: Study the usability perspective of mutation testing by involving real developers.**

  Based on Chapter 4, numerous studies have investigated how useful mutants are. Example studies include *mutant subsumption* [219], *stubborn* mutants [373], and *real-fault coupling* [201, 295]. These studies concern *mutant utility* [203], the efficacy of mutation testing. However, there are few works involving real developers in the empirical studies. We suggest a developer-centric study on mutation testing as it would allow to gain valuable insights on which mutants are found to be most useful.

- **R4: More attentions should be paid on domain-specific mutation testing.**

  In Chapter 5 and Chapter 6, we have designed domain-specific mutation operators for physical computing systems and GPU programs. Compared to conventional mutation operators, we found domain-specific mutation operators to be more helpful in guiding the developers to locate the weaknesses of the programs and write better tests. We would like to encourage researchers to explore more application domains for mutation testing (such as intelligent agents).

References

[1] Antlr. http://www.antlr.org/. [Online; accessed 18-September-2019].

[2] Apache Commons BCEL. https://commons.apache.org/proper/commons-bcel/. [Online; accessed 18-September-2019].

[3] Available mutation operations (PIT). http://pitest.org/quickstart/mutators/. [Online; accessed 10-August-2016].

[4] Code coverage using NVCC compiler. https://devtalk.nvidia.com/default/topic/980545/code-coverage-using-nvcc-compiler/. [Online; accessed 09-October-2019].

[5] Coverage.py. https://https://coverage.readthedocs.io. [Online; accessed 30-January-2018].

[6] CUDA Developer SDK Code Samples. https://www.nvidia.com/object/cuda_get_samples_3.html. [Online; accessed 03-June-2019].

[7] doctest. https://docs.python.org/3/library/doctest.html. [Online; accessed 30-October-2017].

[8] GenMutants. http://pexase.codeplex.com/SourceControl/latest#Projects/GenMutants/GenMutants/Program.cs. [Online; accessed 21-August-2017].

[9] GitHub Repository for Javalanche. https://github.com/david-schuler/javalanche. [Online; accessed 18-April-2018].

[10] GitHub Repository for Milu. https://github.com/yuejia/Milu/tree/develop/src/mutators. [Online; accessed 21-August-2017].

[11] GitHub Repository for MuJava. https://github.com/jeffoffutt/muJava. [Online; accessed 18-April-2018].

[12] GitHub Repository for Mull. https://github.com/mull-project/mull. [Online; accessed 18-September-2019].

[13] GitHub Repository for PIT. https://github.com/hcoles/pitest. [Online; accessed 18-July-2019].

[14] GitHub Repository for Proteum. https://github.com/magsilva/proteum. [Online; accessed 18-April-2018].

[15] GitHub Repository for Proteum/IM. https://github.com/jacksonpradolima/proteumIM2.0. [Online; accessed 18-April-2018].

[16] HCSR04 Manual. https://www.linuxnorth.org/raspi-sump/HC-SR04Users_Manual.pdf. [Online; accessed 30-January-2018].

[17] java-callgraph GitHub Repositry. https://github.com/gousiosg/java-callgraph. [Online; accessed 18-September-2019].

[18] JHawk. http://www.virtualmachinery.com/jhawkprod.htm. [Online; accessed 18-September-2019].

[19] Jumble. http://jumble.sourceforge.net/mutations.html. [Online; accessed 21-August-2017].

[20] LCHB-100 H-bridge. https://www.robotshop.com/media/files/pdf/lchb-100.pdf. [Online; accessed 27-February-2018].

[21] Major v1.3.2. http://mutation-testing.org/downloads/files/major-1.3.2_jre7.zip. [Online; accessed 18-April-2018].

[22] Mutation Testing and Error Seeding-White Box Testing Techniques. http://www.softwaretestinggenius.com/mutation-testing-and-error-seeding-white-box-testing-techniques. [Online; accessed 28-July-2016].

[23] Mutation testing literature survey metadata. https://zenodo.org/badge/latestdoi/95541866. [Online; accessed 17-August-2017].

[24] Mutation testing systems for Java compared. http://pitest.org/java_mutation_testing_systems/. [Online; accessed 18-September-2019].

[25] OpenFoodFacts API. https://world.openfoodfacts.org/. [Online; accessed 31-October-2017].

[26] PexMutator. http://pexase.codeplex.com/SourceControl/latest#Projects/PexMutator/PexMutator/Program.cs. [Online; accessed 21-August-2017].

[27] pytest. https://docs.pytest.org/en/latest/. [Online; accessed 30-October-2017].

[28] Raspberry Camera Module V2. https://www.raspberrypi.org/products/camera-module-v2/. [Online; accessed 31-October-2017].

[29] Raspberry Pi Zero W. https://www.raspberrypi.org/products/raspberry-pi-zero-w/. [Online; accessed 31-October-2017].

[30] SO14057678: cuda matrix multiplication by columns. https://stackoverflow.com/questions/14057678/cuda-matrix-multiplication-by-columns. [Online; accessed 18-March-2019].

[31] SO21677559: array operation using CUDA kernel. https://stackoverflow.com/questions/21677559/array-operation-using-cuda-kernel. [Online; accessed 18-March-2019].

[32] SO25255699: Share memory in CUDA ? How does it CODE work? https://stackoverflow.com/questions/25255699/share-memory-in-cuda-how-does-it-code-work. [Online; accessed 18-March-2019].

[33] SO29158775: My cuda kernal copying matrix with adjustment is not working. https://stackoverflow.com/questions/29158775/my-cuda-kernal-copying-matrix-with-adjustment-is-not-working. [Online; accessed 18-March-2019].

[34] SO29233426: Cuda shared memory bug. https://stackoverflow.com/questions/29233426/cuda-shared-memory-bug. [Online; accessed 18-March-2019].

[35] SO33159171: CUDA C sum 1 dimension of 2D array and return. https://stackoverflow.com/questions/33159171/cuda-c-sum-1-dimension-of-2d-array-and-return. [Online; accessed 18-March-2019].

[36] SO9488590: Shared memory mutex with CUDA - adding to a list of items. https://stackoverflow.com/questions/9488590/shared-memory-mutex-with-cuda-adding-to-a-list-of-items. [Online; accessed 18-March-2019].

[37] SO9859456: cuda thread indexing. https://stackoverflow.com/questions/9859456/cuda-thread-indexing. [Online; accessed 18-March-2019].

[38] Sofya. http://sofya.unl.edu/doc/manual/user/mutator.html. [Online; accessed 21-August-2017].

[39] The Major mutation framework. http://mutation-testing.org/doc/major.pdf. [Online; accessed 21-August-2017].

[40] unittest. https://docs.python.org/3/library/unittest.html. [Online; accessed 30-October-2017].

[41] Allen Troy Acree Jr. On mutation. Technical report, DTIC Document, 1980.

[42] Konstantinos Adamopoulos, Mark Harman, and Robert M Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and evolutionary computation conference*, pages 1338–1349. Springer, 2004.

[43] Hiralal Agrawal, Richard DeMillo, R_ Hathaway, William Hsu, Wynne Hsu, Edward Krauser, Rhonda J Martin, Aditya Mathur, and Eugene Spafford. Design of mutant operators for the c programming language. Technical report, Technical Report

SERC-TR-41-P, Software Engineering Research Center, Department of Computer Science, Purdue University, Indiana, 1989.

[44] Bernhard K Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. Killing strategies for model-based mutation testing. *Software Testing, Verification and Reliability*, 25(8):716–748, 2015.

[45] Roger T Alexander, James M Bieman, Sudipto Ghosh, and Bixia Ji. Mutation of java objects. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 341–351. IEEE, 2002.

[46] Paul Ammann, Marcio Eduardo Delamaro, and Jeff Offutt. Establishing theoretical minimal sets of mutants. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 21–30. IEEE, 2014.

[47] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[48] Paul Ammann and Jeff Offutt. *Introduction to Software Testing, 2nd edition*. Cambridge University Press, 2017.

[49] Paul Anderson. Finding Getters and Setters with Java Reflection. http://asgteach.com/2012/11/finding-getters-and-setters-with-java-reflection/, November 2012. [Online; accessed 18-September-2019].

[50] James H Andrews, Lionel C Briand, and Yvan Labiche. Is mutation an appropriate tool for testing experiments? In *International Conference on Software Engineering*, pages 402–411. IEEE, 2005.

[51] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[52] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M Hierons, and Mark Harman. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *Proceedings of the 36th International Conference on Software Engineering*, pages 573–583. ACM, 2014.

[53] Giuliano Antoniol, Lionel C Briand, M Di Penta, and Yvan Labiche. A case study using the round-trip strategy for state-based class testing. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 269–279. IEEE, 2002.

[54] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 265–275. ACM, 2011.

[55] Arduino. Arduino Mega ADK. https://store.arduino.cc/arduino-mega-adk-rev3. [Online; accessed 27-February-2018].

[56] Arduino. Arduino Uno. https://store.arduino.cc/arduino-uno-rev3. [Online; accessed 25-September-2017].

[57] Erik Arisholm and Lionel C Briand. Predicting fault-prone components in a java legacy system. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 8–17. ACM, 2006.

[58] Jean Arlat, Yves Crouzet, Johan Karlsson, Peter Folkesson, Emmerich Fuchs, and Günther H Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, 2003.

[59] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[60] Al Audet. hcsr04sensor. https://github.com/alaudet/hcsr04sensor. [Online; accessed 30-January-2018].

[61] Simon Außerlechner, Sandra Fruhmann, Wolfgang Wieser, Birgit Hofer, Raphael Spörk, Clemens Mühlbacher, and Franz Wotawa. The right choice matters! smt solving substantially improves model-based debugging of spreadsheets. In *2013 13th International Conference on Quality Software*, pages 139–148. IEEE, 2013.

[62] Richard Baker and Ibrahim Habli. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. *IEEE Transactions on Software Engineering*, 39(6):787–805, 2013.

[63] Douglas Baldwin and Frederick Sayward. Heuristics for determining equivalence of program mutations. Technical report, DTIC Document, 1979.

[64] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.

[65] Ellen Francine Barbosa, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[66] Danilo Bargen. RPLCD. https://github.com/dbrgn/RPLCD. [Online; accessed 30-January-2018].

[67] Benoit Baudry, Franck Fleurey, J-M Jézéquel, and Yves Le Traon. Genes and bacteria for automatic test cases optimization in the. net environment. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 195–206. IEEE, 2002.

[68] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing. *Software Testing, Verification and Reliability*, 15(2):73–96, 2005.

[69] Benoit Baudry, Vu Le Hanh, J-M Jézéquel, and Yves Le Traon. Building trust into oo components using a genetic analogy. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pages 4–14. IEEE, 2000.

[70] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, pages 75–88, 1999.

[71] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer testing in the IDE: Patterns, beliefs, and behavior. *IEEE Transactions on Software Engineering (TSE)*, 45(3):261–284, 2019.

[72] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how and why developers (do not) test in their IDEs. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 179–190. ACM, 2015.

[73] Moritz Beller, Georgios Gousios, and Andy Zaidman. How (much) do developers test? In *Proceedings of the International Conference on Software Engineering (ICSE - volume 2)*, pages 559–562. IEEE, 2015.

[74] Moritz Beller, Niels Spruit, Diomidis Spinellis, and Andy Zaidman. On the dichotomy of debugging behavior among programmers. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 572–583. ACM, 2018.

[75] Fevzi Belli and Mutlu Beyazıt. Exploiting model morphology for event-based testing. *IEEE Transactions on Software Engineering*, 41(2):113–134, 2015.

[76] Fevzi Belli, Mutlu Beyazit, Tomohiko Takagi, and Zengo Furukawa. Mutation testing of" go-back" functions based on pushdown automata. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 249–258. IEEE, 2011.

[77] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[78] Adam Betts, Nathan Chong, Alastair Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for gpu kernels. In *ACM SIGPLAN Notices*, volume 47, pages 113–132. ACM, 2012.

[79] James M Bieman, Sudipto Ghosh, and Roger T Alexander. A technique for mutation of java objects. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on*, pages 337–340. IEEE, 2001.

[80] Robert Binder. *Testing object-oriented systems: models, patterns, and tools.* Addison-Wesley Professional, 2000.

[81] Grady Booch. *Object oriented analysis & design with application.* Pearson Education India, 2006.

[82] Michael Boyer, Kevin Skadron, and Westley Weimer. Automated dynamic analysis of cuda programs. In *Third Workshop on Software Tools for MultiCore Systems*, page 33, 2008.

[83] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[84] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[85] Pearl Brereton, Barbara A Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of systems and software*, 80(4):571–583, 2007.

[86] Lionel C Briand, Yvan Labiche, and Q Lin. Improving statechart testing criteria using data flow information. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.

[87] Lionel C Briand, Yvan Labiche, and Yihong Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *Proceedings of the 26th International Conference on Software Engineering*, pages 86–95. IEEE Computer Society, 2004.

[88] David Bingham Brown, Michael Vaughn, Ben Liblit, and Thomas Reps. The care and feeding of wild-caught mutants. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 511–522. ACM, 2017.

[89] Magiel Bruntink and Arie van Deursen. An empirical study into class testability. *Journal of systems and software*, 79(9):1219–1232, 2006.

[90] Tim Budd and Fred Sayward. Users guide to the pilot mutation system. *Yale University, New Haven, Connecticut, Technique Report*, 114, 1977.

[91] Timothy A Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[92] Timothy A Budd, Richard J Lipton, Richard A DeMillo, and Frederick G Sayward. *Mutation analysis*. Yale University, Department of Computer Science, 1979.

[93] Timothy Alan Budd. *Mutation analysis of program test data*. PhD thesis, Yale University, New Haven, CT, USA, 1980.

[94] Matteo Cargnelutti. Jean-Pierre. `https://github.com/matteocargnelutti/jean-pierre`. [Online; accessed 31-January-2018].

[95] Heung Seok Chae, Gyun Woo, Tae Yeon Kim, Jung Ho Bae, and Won-Young Kim. An automated approach to reducing test suites for testing retargeted c compilers for embedded systems. *Journal of Systems and Software*, 84(12):2053–2064, 2011.

[96] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé Bissyandé, Yves Le Traon, and Koushik Sen. Selecting fault revealing mutants. *arXiv preprint arXiv:1803.07901*, 2018.

[97] Wei Chen, Roland H Untch, Gregg Rothermel, Sebastian Elbaum, and Jeffery Von Ronne. Can fault-exposure-potential estimates improve the fault detection abilities of test suites? *Software Testing, Verification and Reliability*, 12(4):197–218, 2002.

[98] John A Clark, Haitao Dan, and Robert M Hierons. Semantic mutation testing. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 100–109. IEEE, 2010.

[99] Henry Coles. GitHub Repository for PIT. https://github.com/hcoles/pitest. [Online; accessed 18-September-2019].

[100] Henry Coles. Mutation testing - a practitioners perspective. https://github.com/hcoles/slides/blob/master/slides.pdf. [Online; accessed 18-September-2019].

[101] Henry Coles. PIT Incremental Analysis. http://pitest.org/quickstart/incremental_analysis/. [Online; accessed 18-September-2019].

[102] Henry Coles. PIT Main Page. http://pitest.org/. [Online; accessed 18-September-2019].

[103] Henry Coles. PIT Mutation Operators. http://pitest.org/quickstart/mutators/. [Online; accessed 28-May-2019].

[104] W. J. Conover. *Practical Nonparametric Statistics*. Wiley, third edition, 1998.

[105] Reidar Conradi and Alf Inge Wang. *Empirical methods and studies in software engineering: experiences from ESERNET*, volume 2765. Springer, 2003.

[106] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[107] Hernan Czemerinski, Victor Braberman, and Sebastian Uchitel. Behaviour abstraction coverage as black-box adequacy criteria. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 222–231. IEEE, 2013.

[108] Hernan Czemerinski, Victor Braberman, and Sebastian Uchitel. Behaviour abstraction adequacy criteria for api call protocol testing. *Software Testing, Verification and Reliability*, 2015.

[109] Haitao Dan and Robert M Hierons. Smt-c: A semantic mutation testing tools for c. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 654–663. IEEE, 2012.

[110] Romain Delamare, Benoit Baudry, Sudipto Ghosh, Shashank Gupta, and Yves Le Traon. An approach for testing pointcut descriptors in aspectj. *Software Testing, Verification and Reliability*, 21(3):215–239, 2011.

[111] Romain Delamare, Benoit Baudry, and Yves Le Traon. Ajmutator: a tool for the mutation analysis of aspectj pointcut descriptors. In *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*, pages 200–204. IEEE, 2009.

[112] Marcio Eduardo Delamaro, JC Maidonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, 2001.

[113] Márcio Eduardo Delamaro, José Carlos Maldonado, and AP Mathur. Proteum-a tool for the assessment of test adequacy for c programs users guide. In *PCS*, volume 96, pages 79–95, 1996.

[114] Márcio Eduardo Delamaro, José Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. Proteum/im 2.0: An integrated mutation testing environment. In *Mutation testing for the new century*, pages 91–101. Springer, 2001.

[115] R. A. DeMillo. Test adequacy and program mutation. In *Software Engineering, 1989. 11th International Conference on*, pages 355–356, May 1989.

[116] RA DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Software Eng.*, 17(9):900–910, 1991.

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

[118] Richard A DeMillo and Aditya P Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical report, Technical report, Software Engineering Research Center, Purdue University ..., 1991.

[119] Richard A DeMillo and A Jefferson Offutt. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(2):109–127, 1993.

[120] Giovanni Denaro, Alessandro Margara, Mauro Pezze, and Mattia Vivanti. Dynamic data flow testing of object oriented systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 947–958. IEEE Press, 2015.

[121] Lin Deng, Jeff Offutt, Paul Ammann, and Nariman Mirzaei. Mutation operators for testing android apps. *Information and Software Technology*, 81:154–168, 2017.

[122] Alex Denisov and Stanislav Pankevich. Mull it over: mutation testing based on llvm. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 25–31. IEEE, 2018.

[123] Anna Derezinska. Quality assessment of mutation operators dedicated for c# programs. In *Quality Software, 2006. QSIC 2006. Sixth International Conference on*, pages 227–234. IEEE, 2006.

[124] Edsger Wybe Dijkstra et al. Notes on structured programming, 1970.

[125] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: issues and challenges. In *Advanced Information Networking & Applications*, pages 27–33. IEEE, 2010.

[126] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[127] Hyunsook Do and Gregg Rothermel. A controlled experiment assessing test case prioritization techniques via mutation faults. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 411–420. IEEE, 2005.

[128] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Software Engineering, IEEE Transactions on*, 32(9):733–752, 2006.

[129] Lydie du Bousquet and Michel Delaunay. Towards mutation analysis for lustre programs. *Electronic Notes in Theoretical Computer Science*, 203(4):35–48, 2008.

[130] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, (4):438–444, 1984.

[131] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, 2002.

[132] Michael Ellims, Darrel Ince, and Marian Petre. The csaw c mutation tool: Initial results. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 185–192. IEEE, 2007.

[133] Eduard P Enoiu, Daniel Sundmark, Adnan Čaušević, Robert Feldt, and Paul Pettersson. Mutation-based test generation for plc embedded software using model checking. In *IFIP International Conference on Testing Software and Systems*, pages 155–171. Springer, 2016.

[134] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 47. IEEE Computer Society, 2004.

[135] Bo Fang, Karthik Pattabiraman, Matei Ripeanu, and Sudhanva Gurumurthi. Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 221–230. IEEE, 2014.

[136] Chunrong Fang, Zhenyu Chen, Kun Wu, and Zhihong Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Quality Journal*, 22(2):335–361, 2014.

[137] N Farazmand, R Ubal, and D Kaeli. Statistical fault injection-based avf analysis of a gpu architecture. *Proceedings of SELSE*, 12, 2012.

[138] Michael Feathers. *Working effectively with legacy code*. Prentice Hall Professional, 2004.

[139] Robert Feldt and Ana Magazinius. Validity threats in empirical software engineering research-an initial survey. In *Seke*, pages 374–379, 2010.

[140] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software engineering*, 26(8):797–814, 2000.

[141] Andres Flores and Macario Polo. Testing-based process for component substitutability. *Software Testing, Verification and Reliability*, 22(8):529–561, 2012.

[142] Andres Flores and Macario Polo Usaola. Testing-based assessment process for upgrading component systems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 327–336. IEEE, 2008.

[143] Eibe Frank, Mark A. Hall, and Ian H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, 4 edition, 2016.

[144] Phyllis G Frankl, Stewart N Weiss, and Cang Hu. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[145] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[146] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 178–188. IEEE Press, 2012.

[147] Gordon Fraser and Andrea Arcuri. A Large Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2):8, 2014.

[148] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

[149] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

[150] Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. *Software Engineering, IEEE Transactions on*, 38(2):278–292, 2012.

[151] Roy S. Freedman. Testability of software components. *IEEE transactions on Software Engineering*, 17(6):553–564, 1991.

[152] Emmerich Fuchs. An evaluation of the error detection mechanisms in mars using software-implemented fault injection. *Dependable Computing Conference*, pages 73–90, 1996.

[153] Jerry Gao and M-C Shih. A component testability model for verification and measurement. In *29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, volume 2, pages 211–218. IEEE, 2005.

[154] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization. *Journal of Heuristics*, 15(6):617, 2008.

[155] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. Automated oracle data selection support. *IEEE Transactions on Software Engineering*, 41(11):1119–1137, 2015.

[156] Gregory Gay, Matt Staats, Michael Whalen, and Mats PE Heimdahl. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on*, 41(8):803–819, 2015.

[157] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE transactions on software engineering*, 17(12):1284–1288, 1991.

[158] L Bautista Gomez, Franck Cappello, Luigi Carro, Nathan DeBardeleben, Bo Fang, Sudhanva Gurumurthi, Karthik Pattabiraman, Paolo Rech, and M Sonza Reorda. Gpgpus: how to combine high computational power with high reliability. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 341. European Design and Automation Association, 2014.

[159] John B Goodenough and Susan L Gerhart. Toward a theory of test data selection. *IEEE Transactions on software Engineering*, (2):156–173, 1975.

[160] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *IEEE Transactions on Reliability*, 66(3):854–874, Sept 2017.

[161] Rahul Gopinath, Mohammad Amin Alipour, Iftekhar Ahmed, Carlos Jensen, and Alex Groce. On the limits of mutation reduction strategies. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 511–522. IEEE, 2016.

[162] Rahul Gopinath, Carlos Jensen, and Alex Groce. The theory of composite faults. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 47–57. IEEE, 2017.

[163] Dorothy Graham, Erik Van Veenendaal, and Isabel Evans. *Foundations of software testing: ISTQB certification*. Cengage Learning EMEA, 2008.

[164] Kate Gregory and Ade Miller. *C++ AMP: accelerated massive parallelism with Microsoft Visual C++*. Microsoft Press, 2012.

[165] Bernhard JM Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199. IEEE, 2009.

[166] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of things (iot): A vision, architectural elements, and future directions. *Future generation computer systems*, 29(7):1645–1660, 2013.

[167] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, 2011.

[168] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. on Softw. engineering*, (4):279–290, 1977.

[169] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[170] Le Thi My Hanh, Nguyen Thanh Binh, and Khuat Thanh Tung. Survey on mutation-based test data generation. *International Journal of Electrical and Computer Engineering*, 5(5), 2015.

[171] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):10, 2014.

[172] Dan Hao, Lu Zhang, Xingxia Wu, Hong Mei, and Gregg Rothermel. On-demand test suite reduction. In *Proceedings of the 34th International Conference on Software Engineering*, pages 738–748. IEEE Press, 2012.

[173] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Stephen W Keckler, and Joel Emer. Sassifi: An architecture-level fault injection tool for gpu application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 249–258. IEEE, 2017.

[174] Mark Harman, Rob Hierons, and Sebastian Danicic. The relationship between program dependence and mutation analysis. In *Mutation testing for the new century*, pages 5–13. Springer, 2001.

[175] Nannan He, Philipp Rümmer, and Daniel Kroening. Test-case generation for embedded simulink via formal concept analysis. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 224–229. IEEE, 2011.

[176] Mark Hennessy and James F Power. Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software. *Empirical Software Engineering*, 13(4):343–368, 2008.

[177] Rob Hierons, Mark Harman, and Sebastian Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability*, 9(4):233–262, 1999.

[178] Dennis E Hinkle, William Wiersma, Stephen G Jurs, et al. *Applied statistics for the behavioral sciences*. Houghton Mifflin Boston, 1988.

[179] Birgit Hofer, Alexandre Perez, Rui Abreu, and Franz Wotawa. On the empirical evaluation of similarity coefficients for spreadsheets fault localization. *Automated Software Engineering*, 22(1):47–74, 2015.

[180] Birgit Hofer and Franz Wotawa. Why does my spreadsheet compute wrong values? In *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*, pages 112–121. IEEE, 2014.

[181] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

[182] Shin Hong, Matt Staats, Jaemin Ahn, Moonzoo Kim, and Gregg Rothermel. Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation. *Software Testing, Verification and Reliability*, 25(4):334–370, 2015.

[183] Shin Hong, Matt Staats, Jeongseob Ahn, Moonzoo Kim, and Gregg Rothermel. The impact of concurrent coverage metrics on testing effectiveness. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 232–241. IEEE, 2013.

[184] Shan-Shan Hou, Lu Zhang, Tao Xie, Hong Mei, and Jia-Su Sun. Applying interface-contract mutation in regression testing of component-based software. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 174–183. IEEE, 2007.

[185] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Trans. Software Eng.*, (4):371–379, 1982.

[186] Abdulmalik Humayed, Jingqiang Lin, Fengjun Li, and Bo Luo. Cyber-physical systems security—a survey. *IEEE Internet of Things Journal*, 4(6):1802–1831, 2017.

[187] Chen Huo and James Clause. Interpreting coverage information using direct and indirect coverage. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 234–243. IEEE, 2016.

[188] Shamaila Hussain. Mutation clustering. *Ms. Th., Kings College London, Strand, London*, 2008.

[189] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[190] 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, 2007. TAICPART-MUTATION 2007*, pages 169–175. IEEE, 2007.

[191] IEC ISO. Iso 9126/iso, iec (hrsg.): International standard iso/iec 9126: Information technology-software product evaluation. *Quality Characteristics and Guidelines for their use*, pages 12–15, 1991.

[192] David Jackson and Martin R Woodward. Parallel firm mutation of java programs. In *Mutation testing for the new century*, pages 55–61. Springer, 2001.

[193] Karthick Jayaraman, David Harvison, Vijay Ganesh, and Adam Kiezun. jfuzz: A concolic whitebox fuzzer for java. 2009.

[194] Eunkyoung Jee, Donghwan Shin, Sungdeok Cha, Jang-Soo Lee, and Doo-Hwan Bae. Automated test case generation for fbd programs implementing reactor protection system software. *Software Testing, Verification and Reliability*, 24(8):608–628, 2014.

[195] Changbin Ji, Zhenyu Chen, Baowen Xu, and Zhihong Zhao. A Novel Method of Mutation Clustering Based on Domain Analysis. In *SEKE*, pages 422–425, 2009.

[196] Yue Jia and Mark Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic &amp; Industrial Conference*, pages 94–98. IEEE, 2008.

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

[198] Matthieu Jimenez, Thierry Titcheu Chekam, Maxime Cordy, Mike Papadakis, Marinos Kintis, Yves Le Traon, and Mark Harman. Are mutants really natural? a study on how naturalness helps mutant selection. In *12th International Symposium on Empirical Software Engineering and Measurement (ESEM'18)*, 2018.

[199] Shahnewaz Amin Jolly, Vahid Garousi, and Matt M Eskandar. Automated unit testing of a scada control software: an industrial case study based on action research. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 400–409. IEEE, 2012.

[200] René Just, Michael D Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Int'l Symp. on Software Testing and Analysis (ISSTA)*, pages 315–326. ACM, 2014.

[201] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[202] René Just, Gregory M Kapfhammer, and Franz Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 11–20. IEEE, 2012.

[203] René Just, Bob Kurtz, and Paul Ammann. Inferring mutant utility from program context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 284–294. ACM, 2017.

[204] Rene Just, Franz Schweiggert, and Gregory M Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *Int'l Conf. Automated Softw. Engineering (ASE)*, pages 612–615. IEEE, 2011.

[205] Ghani A. Kanawati, Nasser A. Kanawati, and Jacob A. Abraham. Ferrari: A flexible software-based fault and error injection system. *IEEE Transactions on computers*, 44(2):248–260, 1995.

[206] Upulee Kanewala and James M Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 1–10. IEEE, 2013.

[207] Gregory M Kapfhammer, Phil McMinn, and Chris J Wright. Search-based testing of relational schema integrity constraints across multiple database management systems. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 31–40. IEEE, 2013.

[208] Kalpesh Kapoor. Formal analysis of coupling hypothesis for logical faults. *Innovations in Systems and Software Engineering*, 2(2):80–87, 2006.

[209] Jinhyun Kim, Inhye Kang, Jin-Young Choi, and Insup Lee. Timed and resource-oriented statecharts for embedded software. *IEEE Transactions on Industrial Informatics*, 6(4):568–578, 2010.

[210] Kim N King and A Jefferson Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

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

[212] Barbara Kitchenham. Guidelines for performing systematic literature reviews in software engineering. In *Technical Report EBSE-2007-01*, 2007.

[213] Thomas Knauth, Christof Fetzer, and Pascal Felber. Assertion-driven development: Assessing the quality of contracts using meta-mutations. In *The 4th International Workshop on Mutation Analysis (Mutation 2009)*, pages 182–191. IEEE, 2009.

[214] Ron Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, pages 1137–1145. Montreal, Canada, 1995.

[215] Edward W. Krauser, Aditya P. Mathur, and Vernon J Rego. High performance software testing on simd machines. *IEEE Transactions on Software Engineering*, 17(5):403–423, 1991.

[216] Willibald Krenn, Rupert Schlick, Stefan Tiran, Bernhard Aichernig, Elisabeth Jobstl, and Harald Brandl. Momut:: Uml model-based mutation testing for uml. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–8. IEEE, 2015.

[217] Gunnar Kudrjavets, Nachiappan Nagappan, and Thomas Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *2006 17th International Symposium on Software Reliability Engineering*, pages 204–212. IEEE, 2006.

[218] B. Kurtz, P. Ammann, and J. Offutt. Static analysis of mutant subsumption. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 1–10, April 2015.

[219] Bob Kurtz, Paul Ammann, Marcio E Delamaro, Jeff Offutt, and Lin Deng. Mutant subsumption graphs. In *Software testing, verification and validation workshops (ICSTW), 2014 IEEE seventh international conference on*, pages 176–185. IEEE, 2014.

[220] Bob Kurtz, Paul Ammann, Jeff Offutt, Márcio E Delamaro, Mariet Kurtz, and Nida Gökçe. Analyzing the validity of selective mutation with dominator mutants. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 571–582. ACM, 2016.

[221] Jung-Hyun Kwon, In-Young Ko, Gregg Rothermel, and Matt Staats. Test case prioritization based on information retrieval concepts. In *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, volume 1, pages 19–26. IEEE, 2014.

[222] Lynn Kysh. Difference between a systematic review and a literature review. https://dx.doi.org/10.6084/m9.figshare.766364.v1, 2013. [Online; accessed 4-August-2016].

[223] Kim G Larsen, Marius Mikucionis, Brian Nielsen, and Arne Skou. Testing real-time embedded software using uppaal-tron: an industrial case study. In *Proc. Int'l Conf. on Embedded Software*, pages 299–306. ACM, 2005.

[224] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54–72, 2012.

[225] Chiyoung Lee, Se-Won Kim, and Chuck Yoo. Vadi: Gpu virtualization for an automotive platform. *IEEE Transactions on Industrial Informatics*, 12(1):277–290, 2015.

[226] Alan Leung, Manish Gupta, Yuvraj Agarwal, Rajesh Gupta, Ranjit Jhala, and Sorin Lerner. Verifying gpu kernels by test amplification. In *ACM SIGPLAN Notices*, volume 47, pages 383–394. ACM, 2012.

[227] Guodong Li and Ganesh Gopalakrishnan. Scalable smt-based verification of gpu kernel functions. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196. ACM, 2010.

[228] Nan Li, Upsorn Praphamontripong, and Jeff Offutt. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In *ICST workshops*, pages 220–229. IEEE, 2009.

[229] Ping Li, Toan Huynh, Marek Reformat, and James Miller. A practical approach to testing gui systems. *Empirical Software Engineering*, 12(4):331–357, 2007.

[230] R Lipton. Fault diagnosis of computer programs. *Student Report, Carnegie Mellon University*, 1971.

[231] Ming-Hao Liu, You-Feng Gao, Jin-Hui Shan, Jiang-Hong Liu, Lu Zhang, and Jia-Su Sun. An approach to test data generation for killing multiple mutants. In *Software Maintenance, 2006. ICSM 2006. IEEE International Conference on*, pages 113–122. IEEE, 2006.

[232] Yiling Lou, Dan Hao, and Lu Zhang. Mutation-based test-case prioritization in software evolution. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 46–57. IEEE, 2015.

[233] Heng Lu, WK Chan, and TH Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Int'l Symp. Foundations of Software Engineering (FSE)*, pages 242–252. ACM, 2006.

[234] David Luebke and Greg Humphreys. How gpus work. *Computer*, 40(2):96–100, 2007.

[235] Michael R Lyu, Zubin Huang, Sam KS Sze, and Xia Cai. An empirical study on testing and fault tolerance for software reliability engineering. In *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, pages 119–130. IEEE, 2003.

[236] Yu-Seung Ma and Sang-Woon Kim. Mutation testing cost reduction by clustering overlapped mutants. *J. Systems and Software*, 115:18–30, 2016.

[237] Yu-Seung Ma and Jeff Offutt. Description of class mutation mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep*, 2005.

[238] Yu-Seung Ma and Jeff Offutt. Description of method-level mutation operators for java. *Electronics and Telecommunications Research Institute, Korea, Tech. Rep*, 2005.

[239] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[240] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for java. In *Proceedings of the 28th international conference on Software engineering*, pages 827–830. ACM, 2006.

[241] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Józala. Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation. *Software Engineering, IEEE Transactions on*, 40(1):23–42, 2014.

[242] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 667–676. ACM, 2007.

[243] Pedro Reales Mateo, Macario Polo Usaola, and Jeff Offutt. Mutation at the multi-class and system levels. *Science of Computer Programming*, 78(4):364–387, 2013.

[244] Aditya P Mathur and W Eric Wong. An empirical comparison of data flow and mutation-based test adequacy criteria. *Software Testing, Verification and Reliability*, 4(1):9–31, 1994.

[245] MATLAB. *version 9.3.0 (R2017b)*. The MathWorks Inc., Natick, Massachusetts, 2017.

[246] MATLAB. *version 9.6.0 (R2019a)*. The MathWorks Inc., Natick, Massachusetts, 2019.

[247] Joseph A Maxwell. *Qualitative research design: An interactive approach*, volume 41. Sage publications, 2012.

[248] Paul McGuire. *Getting started with pyparsing*. " O'Reilly Media, Inc.", 2007.

[249] Phil Mcminn, Chris J Wright, and Gregory M Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(1):8, 2015.

[250] Xinxin Mei and Xiaowen Chu. Dissecting gpu memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[251] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78. ACM, 2014.

[252] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. Efficient javascript mutation testing. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 74–83. IEEE, 2013.

[253] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 153–162. IEEE, 2014.

[254] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software evolution*, pages 173–202. Springer, 2008.

[255] Ivan Moore. Jester-a junit test tester. *Proc. of 2nd XP*, pages 84–87, 2001.

[256] Wayne Motycka. Installation Instructions for Sofya. `http://sofya.unl.edu/doc/manual/installation.html`, 7 2013. [Online; accessed 25-August-2016].

[257] Samar Mouchawrab, Lionel C Briand, and Yvan Labiche. A measurement framework for object-oriented software testability. *Information and software technology*, 47(15):979–997, 2005.

[258] Elfurjani S Mresa and Leonardo Bottaci. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing Verification and Reliability*, 9(4):205–232, 1999.

[259] Aaftab Munshi. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–314. IEEE, 2009.

[260] Syed Shariyar Murtaza, Nazim Madhavji, Mechelle Gittens, and Zude Li. Diagnosing new faults using mutants and prior faults (nier track). In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 960–963. IEEE, 2011.

[261] Akbar Siami Namin and James H Andrews. On sufficiency of mutants. In *Software Engineering-Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 73–74. IEEE, 2007.

[262] Akbar Siami Namin and James H Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 57–68. ACM, 2009.

[263] Mohd Nazir, Raees A Khan, and K Mustafa. Testability estimation framework. *International Journal of Computer Applications*, 2(5):9–14, 2010.

[264] Simeon C Ntafos. On testing with required elements. In *Proceedings of COMPSAC*, volume 81, pages 132–139, 1981.

[265] Simeon C Ntafos. An evaluation of required element testing strategies. In *Proceedings of the 7th international conference on Software engineering*, pages 250–256. IEEE Press, 1984.

[266] Simeon C Ntafos. On required element testing. *Software Engineering, IEEE Transactions on*, (6):795–803, 1984.

[267] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2010. Version 3.2.

[268] A Offutt. The coupling effect: fact or fiction. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 131–140. ACM, 1989.

[269] A Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(1):5–20, 1992.

[270] A Jefferson Offutt and W Michael Craft. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability*, 4(3):131–154, 1994.

[271] 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.

[272] A Jefferson Offutt and Stephen D Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, 1994.

[273] A Jefferson Offutt and Shaoying Liu. Generating test data from sofl specifications. *Journal of Systems and Software*, 49(1):49–62, 1999.

[274] A Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software testing, verification and reliability*, 7(3):165–192, 1997.

[275] A Jefferson Offutt, Jie Pan, and Jeffrey M Voas. Procedures for reducing the size of coverage-based test sets. In *Proceedings of the Twelfth International Conference on Testing Computer Software*. Citeseer, 1995.

[276] A Jefferson Offutt, Roy P Pargas, Scott V Fichter, and Prashant K Khambekar. Mutation testing of software using a mimd computer. In *in 1992 International Conference on Parallel Processing*. Citeseer, 1992.

[277] A Jefferson Offutt and Roland H Untch. Mutation 2000: Uniting the orthogonal. In *Mutation testing for the new century*, pages 34–44. Springer, 2001.

[278] Andrew Jefferson Offutt. *Automatic test data generation*. PhD thesis, Georgia Institute of Technology, 1988.

[279] Jeff Offutt. A mutation carol: Past, present and future. *Information and Software Technology*, 53(10):1098–1107, 2011.

[280] Younju Oh, Junbeom Yoo, Sungdeok Cha, and Han Seong Son. Software safety analysis of function block diagrams using fault trees. *Reliability Engineering & System Safety*, 88(3):215–228, 2005.

[281] Open Source Robotics Foundation. Robot Operating System. http://www.ros.org/, 10 2014. [Online; accessed 16-February-2018].

[282] Dan O'Sullivan and Tom Igoe. *Physical computing: sensing and controlling the physical world with computers*. Course Technology Press, 2004.

[283] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. 2008.

[284] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 130–141. ACM, 2016.

[285] A. Panichella, F. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.

[286] Annibale Panichella and Urko Rueda Molina. Java unit testing tool competition - fifth round. In *10th IEEE/ACM International Workshop on Search-Based Software Testing (SBST)*, pages 32–38, 2017.

[287] Mike Papadakis, Christopher Henard, and Yves Le Traon. Sampling program inputs with mutation analysis: Going beyond combinatorial interaction testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*, pages 1–10. IEEE, 2014.

[288] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 936–946. IEEE, 2015.

[289] 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.

[290] Mike Papadakis and Yves Le Traon. Using mutants to locate "unknown" faults. In *7th International Workshop on Mutation Analysis (MUTATION'12)*, pages 691–700. IEEE, 2012.

[291] Mike Papadakis and Yves Le Traon. Metallaxis-fl: mutation-based fault localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[292] Mike Papadakis and Nicos Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Software reliability engineering (ISSRE), 2010 IEEE 21st international symposium on*, pages 121–130. IEEE, 2010.

[293] Mike Papadakis and Nicos Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal*, 19(4):691–723, 2011.

[294] Mike Papadakis and Nicos Malevris. Mutation based test case generation via a path selection strategy. *Information and Software Technology*, 54(9):915–932, 2012.

[295] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults. In *40th International Conference on Software Engineering, May 27-3 June 2018, Gothenburg, Sweden*, 2018.

[296] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[297] Goran Petrovic and Marko Ivankovic. State of mutation testing at Google. In *Proceedings of the International Conference on Software Engineering in Practice (ICSE SEIP)*, 2018.

[298] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICST Workshops)*, pages 47–53. IEEE, 2018.

[299] Yuhua Qi, Xiaoguang Mao, and Yan Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE, 2013.

[300] Xiao Qu, Myra B Cohen, and Gregg Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 75–86. ACM, 2008.

[301] Ross Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann, San Mateo, CA, 1993.

[302] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Design Automation Conference*, pages 731–736. IEEE, 2010.

[303] Paolo Rech, Laércio Lima Pilla, Philippe Olivier Alexandre Navaux, and Luigi Carro. Impact of gpus parallelism management on safety-critical and hpc applications reliability. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 455–466. IEEE, 2014.

[304] Alice Richardson. Nonparametric statistics for non-statisticians: A step-by-step approach by gregory w. corder, dale i. foreman. *International Statistical Review*, 78(3):451–452, 2010.

[305] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.

[306] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*, pages 179–188. IEEE, 1999.

[307] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, 2001.

[308] RPi-Distro. GPIO Zero. https://github.com/RPi-Distro/python-gpiozero. [Online; accessed 30-January-2018].

[309] Vladimir V Rubanov and Eugene A Shatokhin. Runtime verification of linux kernel modules based on call interception. In *Int'l Conf. Software Testing, Verification and Validation (ICST)*, pages 180–189. IEEE, 2011.

[310] Matthew J Rutherford, Antonio Carzaniga, and Alexander L Wolf. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *Software Engineering, IEEE Transactions on*, 34(4):452–470, 2008.

[311] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[312] David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 69–80. ACM, 2009.

[313] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for Java. In *Proc. ESEC/FSE*, pages 297–298. ACM, 2009.

[314] David Schuler and Andreas Zeller. (un-) covering equivalent mutants. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 45–54. IEEE, 2010.

[315] David Schuler and Andreas Zeller. Checked coverage: an indicator for oracle quality. *Software Testing, Verification and Reliability*, 23(7):531–551, 2013.

[316] Zary Segall, D Vrsalovic, D Siewiorek, D Ysskin, J Kownacki, J Barton, R Dancey, A Robinson, and T Lin. Fiat-fault injection based automated testing environment. In *Proc. 18th Int. Symposium on Fault-Tolerant Computing*, page 394. IEEE, 1988.

[317] Hossain Shahriar and Mohammad Zulkernine. Music: Mutation-based sql injection vulnerability checking. In *Quality Software, 2008. QSIC'08. The Eighth International Conference on*, pages 77–86. IEEE, 2008.

[318] Hossain Shahriar and Mohammad Zulkernine. Mutation-based testing of format string bugs. In *High Assurance Systems Engineering Symposium, 2008. HASE 2008. 11th IEEE*, pages 229–238. IEEE, 2008.

[319] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. Balancing trade-offs in test-suite reduction. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 246–256. ACM, 2014.

[320] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. Comparing and combining test-suite reduction and regression test selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 237–247. ACM, 2015.

[321] Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae. Empirical evaluation on fbd model-based test coverage criteria using mutation analysis. In *International Conference on Model Driven Engineering Languages and Systems*, pages 465–479. Springer, 2012.

[322] Akbar Siami Namin, James H Andrews, and Duncan J Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*, pages 351–360. ACM, 2008.

[323] Y Singh and A Saha. Predicting testability of eclipse: a case study. *Journal of Software Engineering*, 4(2):122–136, 2010.

[324] Ben H Smith and Laurie Williams. On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, 14(3):341–369, 2009.

[325] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pages 12–23. IEEE, 2018.

[326] Matt Staats, Gregory Gay, and Mats PE Heimdahl. Automated oracle creation support, or: how i learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering*, pages 870–880. IEEE Press, 2012.

[327] Matt Staats, Pablo Loyola, and Gregg Rothermel. Oracle-centric test case prioritization. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 311–320. IEEE, 2012.

[328] Matt Staats, Michael W Whalen, and Mats PE Heimdahl. Better testing through oracle selection (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 892–895. ACM, 2011.

[329] J A Stankovic, I Lee, A Mok, and R Rajkumar. Opportunities and obligations for physical computing systems. *Computer*, 38(11):23–31, 2005.

[330] Matthew Stephan, Manar H Alalfi, and James R Cordy. Towards a taxonomy for Simulink model mutations. In *Software Testing, Verification and Validation Workshops (ICSTW)*, pages 206–215. IEEE, 2014.

[331] Matthew Stephan and James R Cordy. Model clone detector evaluation using mutation analysis. In *ICSME*, pages 633–638, 2014.

[332] Sam S Stone, Justin P Haldar, Stephanie C Tsao, BP Sutton, Z-P Liang, et al. Accelerating advanced mri reconstructions on gpus. *Journal of parallel and distributed computing*, 68(10):1307–1318, 2008.

[333] Joanna Strug and Barbara Strug. Machine learning approach in mutation testing. In *IFIP International Conference on Testing Software and Systems*, pages 200–214. Springer, 2012.

[334] Ahyoung Sung, Byoungju Choi, and Seokkyoo Shin. An interface test model for hardware-dependent software and embedded os api of the embedded system. *Computer Standards & Interfaces*, 29(4):430–443, 2007.

[335] Pushpa R Suri and Harsha Singhani. Object oriented software testability survey at designing and implementation phase. *International Journal of Science and Research*, 4(4):3047–3053, 2015.

[336] Hiroyuki Takizawa and Hiroaki Kobayashi. Hierarchical parallel processing of large scale data clustering on a pc cluster with gpu co-processing. *The Journal of Supercomputing*, 36(3):219–234, 2006.

[337] Texas Instruments. 74HC74 D flip-flop Data Sheet. `http://www.utm.edu/staff/leeb/logic/74ls74.pdf`. [Online; accessed 14-June-2017].

[338] Texas Instruments. 74LS107 JK flip-flop Data Sheet. `http://www.utm.edu/staff/leeb/logic/74ls107.pdf`. [Online; accessed 14-June-2017].

[339] Sameer Tilak, Nael B Abu-Ghazaleh, and Wendi Heinzelman. A taxonomy of wireless micro-sensor network models. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(2):28–36, 2002.

[340] Ayse Tosun and Ayse Bener. Reducing false alarms in software defect prediction by decision threshold optimization. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 477–480. IEEE Computer Society, 2009.

[341] Wei-Tek Tsai, Lian Yu, Feng Zhu, and Ray Paul. Rapid embedded system testing using verification patterns. *IEEE software*, pages 68–75, 2005.

[342] Javier Tuya, Ma José Suárez-Cabal, and Claudio De La Riva. Mutating database queries. *Information and Software Technology*, 49(4):398–417, 2007.

[343] Javier Tuya, María José Suárez-Cabal, and Claudio De La Riva. Full predicate coverage for testing sql database queries. *Software Testing, Verification and Reliability*, 20(3):237–288, 2010.

[344] Roland Untch, A Jefferson Offutt, and Mary Jean Harrold. Mutation testing using mutant schemata. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 139–148, 1993.

[345] Roland H Untch. Mutation-based software testing using program schemata. In *Proceedings of the 30th annual Southeast regional conference*, pages 285–291. ACM, 1992.

[346] Arie van Deursen. Software Testing in 2048. https://speakerdeck.com/avandeursen/software-testing-in-2048, 1 2016. [Online; accessed 13-Sep-2016].

[347] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, pages 1–31, 2017.

[348] P Vilela, M Machado, and WE Wong. Testing for security vulnerabilities in software. *Software Engineering and Applications*, 2002.

[349] Auri Marcelo Rizzo Vincenzi, Jose Carlos Maldonado, Ellen Francine Barbosa, and Marcio Eduardo Delamaro. Unit and integration testing strategies for c programs using mutation. *Software Testing, Verification and Reliability*, 11(4):249–268, 2001.

[350] Auri Marcelo Rizzo Vincenzi, Elisa Yumi Nakagawa, José Carlos Maldonado, Márcio Eduardo Delamaro, and Roseli Aparecida Francelin Romero. Bayesian-learning based guidelines to determine equivalent mutants. *International Journal of Software Engineering and Knowledge Engineering*, 12(06):675–689, 2002.

[351] Willem Visser. What makes killing a mutant hard. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 39–44. ACM, 2016.

[352] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, pages 203–232, 2003.

[353] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. Search-based data-flow test generation. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 370–379. IEEE, 2013.

[354] Jeffrey M. Voas. Pie: A dynamic failure-based technique. *IEEE Transactions on software Engineering*, 18(8):717–727, 1992.

[355] Jeffrey M. Voas and Keith W Miller. Software testability: The new verification. *IEEE software*, 12(3):17–28, 1995.

[356] Anneliese von Mayrhauser, Michael Scheetz, Eric Dahlman, and Adele E Howe. Planner based error recovery testing. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pages 186–195. IEEE, 2000.

[357] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proc. of the Joint Meeting of the European Software Engineering Conference and the Int'l Symp. on Software Engineering (ESEC/FSE)*, pages 205–214. ACM, 2007.

[358] KS Wah. Fault coupling in finite bijective functions. *Software Testing, Verification and Reliability*, 5(1):3–47, 1995.

[359] KS Wah. A theoretical study of fault coupling. *Software testing, verification and reliability*, 10(1):3–45, 2000.

[360] KS How Tai Wah. An analysis of the coupling effect i: single test data. *Science of Computer Programming*, 48(2):119–161, 2003.

[361] Xinming Wang, Shing-Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 45–55. IEEE Computer Society, 2009.

[362] Michael Whalen, Gregory Gay, Dongjiang You, Mats PE Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 102–111. IEEE, 2013.

[363] David A. Wheeler. SLOCCount. `https://www.dwheeler.com/sloccount/`. [Online; accessed 13-March-2019].

[364] Rudolf Wille. Formal concept analysis as mathematical theory of concepts and concept hierarchies. In *Formal concept analysis*, pages 1–33. Springer, 2005.

[365] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 38. ACM, 2014.

[366] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[367] W Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[368] Martin R Woodward. Mutation testing-an evolving technique. In *Software Testing for Critical Systems, IEE Colloquium on*, pages 3–1. IET, 1990.

[369] Martin R. Woodward, Michael A. Hennell, and David Hedley. A measure of control flow complexity in program text. *IEEE Transactions on Software Engineering*, (1):45–50, 1979.

[370] Christopher Wright. *Mutation analysis of relational database schemas*. PhD thesis, University of Sheffield, 2015.

[371] Xiaoyuan Xie, Joshua WK Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

[372] Dianxiang Xu and Junhua Ding. Prioritizing state-based aspect tests. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 265–274. IEEE, 2010.

[373] Xiangjuan Yao, Mark Harman, and Yue Jia. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering*, pages 919–930. ACM, 2014.

[374] Keun Soo Yim, Cuong Pham, Mushfiq Saleheen, Zbigniew Kalbarczyk, and Ravishankar Iyer. Hauberk: Lightweight silent data corruption error detector for gpgpu. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pages 287–300. IEEE, 2011.

[375] Hoijin Yoon and Byoungju Choi. Effective test case selection for component customization and its application to enterprise javabeans. *Software Testing, Verification and Reliability*, 14(1):45–70, 2004.

[376] Yuan Zhan and John A Clark. Search-based mutation testing for simulink models. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2005.

[377] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.

[378] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang. Predictive mutation testing. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[379] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. Predictive mutation testing. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 342–353. ACM, 2016.

[380] Lingming Zhang, Darko Marinov, and Sarfraz Khurshid. Faster mutation testing inspired by test prioritization and reduction. In *Proc. Int'l. Symp. on Software Testing and Analysis (ISSTA)*, pages 235–245. ACM, 2013.

[381] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. An empirical study of junit test-suite reduction. In *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pages 170–179. IEEE, 2011.

[382] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan De Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010.

[383] Qiushuang Zhang and Ian G Harris. A data flow fault coverage metric for validation of behavioral hdl descriptions. In *Proc. Int'l Conf on Computer-aided design*, pages 369–373. IEEE, 2000.

[384] Yucheng Zhang and Ali Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 214–224. ACM, 2015.

[385] YuMing Zhou, Hareton Leung, QinBao Song, JianJun Zhao, HongMin Lu, Lin Chen, and BaoWen Xu. An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. *Science china information sciences*, 55(12):2800–2815, 2012.

[386] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

[387] Qianqian Zhu. Mutation testing Tools. https://zenodo.org/badge/latestdoi/122769075. [Online; accessed 24-February-2018].

[388] Qianqian Zhu. MutPhy GitHub Repository. https://zenodo.org/badge/latestdoi/136980504. [Online; accessed 11-June-2018].

[389] Qianqian Zhu. Replicate Package. https://doi.org/10.5281/zenodo.3484715. [Online; accessed 16-December-2019].

[390] Qianqian Zhu. GitHub Repository for Mutation Observer. https://zenodo.org/badge/latestdoi/147203995, 2019. [Online; accessed 18-September-2019].

[391] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. Speeding-up mutation testing via data compression and state infection. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 103–109. IEEE, 2017.

[392] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. An investigation of compression techniques to speed up mutation testing. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 274–284. IEEE, 2018.

[393] Qianqian Zhu, Annibale Panichella, and Andy Zaidman. A systematic literature review of how mutation testing supports quality assurance processes. *Software Testing, Verification and Reliability*, 28(6):e1675, 2018. e1675 stvr.1675.

[394] Qianqian Zhu and Andy Zaidman. Mutation testing for physical computing. In *2018 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 289–300. IEEE, 2018.

[395] Qianqian Zhu and Andy Zaidman. Massively parallel, highly efficient, but what about the test suite quality? applying mutation testing to gpu programs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 209–219. IEEE, 2020.

Curriculum Vitæ

Qianqian Zhu

23-12-1990	Born in Wenzhou, Zhejiang Province, China.

Education

2015.11-2019.11	**Ph.D. in Software Engineering** *Delft University of Technology*, Netherlands *Promotors:* Prof. Dr. Andy Zaidman & Prof. Dr. Arie van Deursen *Co-promotor:* dr. Annibale Panichella
2013.10-2014.10	**M.Sc. in Computer Science** *Imperial College London*, UK *Grade:* Distinction & TOP 3rd
2009.09-2013.06	**B.Eng. in Biosystems Engineering** *Zhejiang University*, China *Grade:* 4.11/5.0 & TOP 10%

Experience

2015-2019	**Reviewer** JSS, STVR, ICST, ICSE, SCAM, SANER
2019, 2020	**Program committee member** Mutation Testing Workshop
2018	**Web chair** The 15th International Conference on Mining Software Repositories (MSR 2018)
2017.04-2017.07	**Teaching Assistant** Software Testing and Quality Engineering, prof. dr. A. van Deursen
2014.11-2015.02	**Data Scientist** *Go Capture*, Shanghai, China
2009-2010	**Steering committee member** College Students Association Union of Zhejiang University

List of Publications

5. **Qianqian Zhu** and Andy Zaidman, *Massively Parallel, Highly Efficient, but What About the Test Suite Quality? Applying Mutation Testing to GPU Programs*, Proceedings of the 11th International Conference on Software Testing, Verification, and Validation (ICST), Porto, Portugal, 2020

4. **Qianqian Zhu** and Andy Zaidman, *Mutation Testing for Physical Computing*, The 18th IEEE International Conference on Software Quality, Reliability, and Security (QRS), Lisbon, Portugal, 2018.

3. **Qianqian Zhu**, Annibale Panichella and Andy Zaidman, *An Investigation of Compression Techniques to Speed up Mutation Testing*, Proceedings of the 11th International Conference on Software Testing, Verification, and Validation (ICST), Västerås, Sweden, 2018.

2. **Qianqian Zhu**, Annibale Panichella and Andy Zaidman, *Speeding-Up Mutation Testing via Data Compression and State Infection*, 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Tokyo, 2017, pp. 103-109.

1. **Qianqian Zhu**, Annibale Panichella and Andy Zaidman, *A systematic literature review of how mutation testing supports quality assurance processes*, Software Testing Verification Reliability 2018;28:e1675.

THE APPLICATION

PERSPECTIVE OF MUTATION TESTING