Project on

Mutation Testing

Authors
Prabin Kumar Shrestha, *pkshrestha@mun.ca*
Daniel Manasseh Ofei Kwafo, *dmokwafo@mun.ca*

ENGI 9839 - SOFTWARE VERIFICATION AND VALIDATION

MASC. IN SOFTWARE ENGINEERING
MEMORIAL UNIVERSITY OF NEWFOUNDLAND

July 2023

1 Introduction

Mutation testing is a fault-based testing technique that intentionally introduces defects or mutations into a program under evaluation. By simulating different types of faults, mutation testing aims to uncover potential weaknesses in the program's code and evaluate the effectiveness of the existing test suite in detecting these faults [1]. This technique has gained maturity over time and is increasingly recognized and utilized in both academic and industrial contexts. The evaluation is typically done using a criterion called the "mutation adequacy score," obtained by executing a set of mutated programs known as mutants, which are created by making simple syntactic changes to the original program [2].

The flexibility of mutation testing allows its application in various testing domains, including unit testing, integration testing, specification, design, and system levels. It has been successfully employed in popular programming languages such as C, C++, C#, Java, JavaScript, and Ruby.

Despite its effectiveness in evaluating test set quality, mutation testing encounters several challenges that hinder its practicality as a testing technique. One notable challenge is the significant computational burden imposed by the execution of a vast number of mutants against a given test set, making it a resource-intensive process. Additionally, other issues arise in relation to the human effort required for utilizing mutation testing. Several cost reduction techniques, both mutation reduction and execution cost reduction, have been devised to address the challenges associated with mutation testing. [2]

2 Objective

The goal of this study is to learn in-depth about mutation testing, including its theories, techniques, and applications in different areas. Additionally, we aim to put this knowledge into practice by implementing mutation testing in a specific programming language and using the relevant tools. By doing so, we hope to gain practical experience and understand how effective mutation testing is in finding errors in software. This research will contribute to our understanding of mutation testing and its usefulness in real-world software development.

3 Theory of Mutation Testing

Mutation analysis is a powerful and computationally intensive method for assessing the effectiveness of test cases in detecting faults. It requires repeated execution of multiple faulty versions of the program (mutants) under test. Mutation Testing holds the potential to effectively identify suitable test data capable of detecting genuine faults [3]. Mutation analysis was done in detail in [4]

3.1 Background

The All-Uses Data Flow Criteria is a traditional method, used in software testing, to thoroughly validate the data flow within a program. Its purpose is to ensure that all possible paths in the code are tested, which helps detect potential errors and vulnerabilities. This involves checking the flow of data, including input, processing, and output, across various paths and branches within the software. More information about this criteria, work done by [5], can be found in the work of Mathur and Wang [6].

There exists an extensive array of potential faults for a given program, making it unfeasible to create mutants that encompass all of them. Consequently, conventional Mutation Testing focuses solely on a portion of these faults, specifically those closely resembling the correct program version, under the assumption that they adequately represent all possible faults [2].

In the context of software testing, Mathur and Wang (1995) [6] discuss a method for decreasing the number of mutants that need to be tested. One prevalent approach for achieving this is by randomly selecting a subset of mutants for evaluation. However, this random selection method might not adequately consider the specific fault detection capabilities of different mutant types. To address this limitation, the researchers proposed a technique called ”constrained mutation” or ”selective mutation.” This approach involves limiting the generated mutants to two specific types: absolute value mutants (abs) and mutants obtained by replacing relational operators (ror). The primary goal of this method is to streamline the testing process, thereby reducing the testing workload, while still maintaining the effectiveness of mutation testing in identifying faults. The researchers put forward two versions of the constrained mutation technique. The first variant entails restricting the mutant operators exclusively to abs and ror, while the second variant involves random selection of mutants from each mutant type.

To compare the effectiveness of different testing approaches, the researchers conducted experiments. They specifically compared mutation testing and its variations with the all-uses data flow criteria. In contrast to certain previous studies where human testers created the test sets, Mathur and Wang randomly generated the test sets for their experiments. The results of their experiments confirmed the hypothesis that a testing criterion based on one of the two mutation variants performs better than the all-uses criterion in terms of fault detection effectiveness. Consequently, this suggests that mutation-based approaches are more adept at identifying faults in software systems compared to the all-uses data flow criteria.

3.2 Hypothesis

Yue Jia and Mark Harman [2] mentioned in their paper that traditional mutation Testing focuses on a specific subset of faults, namely those that are in close proximity to the correct version of the program. The underlying assumption is that these selected faults will effectively simulate all possible faults. This theory is built upon two fundamental hypotheses: the Competent Programmer Hypothesis (CPH), and the Coupling Effect [4].

3.2.1 Competent Programmer Hypothesis (CPH)

The Competent Programmer Hypothesis suggests that programmers, in general, are competent and tend to create programs that are close to the correct version. This assumption implies that the code produced by competent programmers might contain faults, but these faults are expected to be simple and easily correctable with minor syntactical changes. In mutation testing, various mutations or changes are applied to the original program to create mutant versions. Each mutant represents a specific fault introduced into the original code. However, according to the CPH, only certain types of faults are considered, which are constructed from simple syntactical changes. These syntactical changes are meant to represent the kinds of mistakes that competent programmers are more likely to make [4] [2].

For mutation analysis to be practical, three factors are crucial [4]:

1. The set of simple mutants should be small.
2. Errors must be reliably detectable by mutation analysis tools.
3. The issue of equivalence between mutants and the original program should be manageable as a small subproblem.

The validity of the competent programmer hypothesis has been uncertain, with conflicting claims in past research [7]. The researchers has offered a new perspective on the competent programmer hypothesis and its connection to mutation testing. The approach involves attempting to recreate real-world bugs by applying chains of mutations to the code. By analyzing the lengths of these mutation paths, they aim to determine whether the source code is indeed almost correct, or if substantial variations are necessary to induce bugs. They found that the competent programmer hypothesis holds true to some extent. However, it appears that the current set of mutation operators may not be sufficient to generate representative real-world bugs accurately.

3.2.2 Coupling Effect

The program may contain complex errors that are not explicit mutants and cannot be detected by the existing test data. In contrast to the CPH (Code-Programmer Hypothesis) which focuses on a programmer's behavior, the Coupling Effect is concerned with the types of faults utilized in mutation analysis [2]. It states that ”Test data on which all simple mutants fail is so sensitive to changes in the program that it is likely that all complex mutants also fail.” [4]

Offcut [8] defines simple fault as correctable with a single change to a source statement, while a complex fault requires multiple changes. Mutations introduce simple faults into a program, and higher-order mutants represent complex faults that result from multiple mutations, forming a subset of the overall complex faults present in the program. Offcut states the coupling effect hypothesis as ”Complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults.” Offcut in his paper [8], has introduced Mutation Coupling Effect Hypothesis concept which states ”Complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will detect a large percentage of the complex mutants.”

The study [8] focused on comparing the effectiveness of test data designed to detect basic mutations (1-order mutants) with their ability to identify more complex mutations (2-order and 3-order mutants). Surprisingly, the test data developed for basic mutations successfully killed a significant number of 2-order mutants, indicating their effectiveness in detecting more complex faults. The surviving 2-order mutants did not exhibit specific characteristics making them difficult to detect, further supporting the reliability of the test data. Moreover, the test data for basic mutations killed a higher percentage of 2-order mutants compared to specific 2-order test data, and a similar trend was observed with 3-order mutants.

Include more if time and space allows

3.3 Process of Mutation Testing

The system is equipped with a test set, denoted as T. Prior to initiating the mutation analysis, it is essential to ensure the successful execution of this test set against the original program, referred to as p, in order to validate its correctness for the given test case. If any inconsistencies or errors are identified in p, they must be resolved before proceeding to evaluate other mutants. Conversely, if p is deemed to be error-free, each mutant, represented as p', will be subjected to execution using the same test set T. If the output obtained from executing p' differs from the output obtained from executing p for any test case within T, the mutant p' is classified as ”killed”. On the other hand, if the obtained results are identical, the mutant is considered to have ”survived” [2]. Certain mutants may be considered syntactically invalid, leading to unsuccessful compilation. These non-viable mutants are commonly known as ”stillborn” mutants and must be excluded from subsequent analysis [1]. The process is shown in the Figure 1.

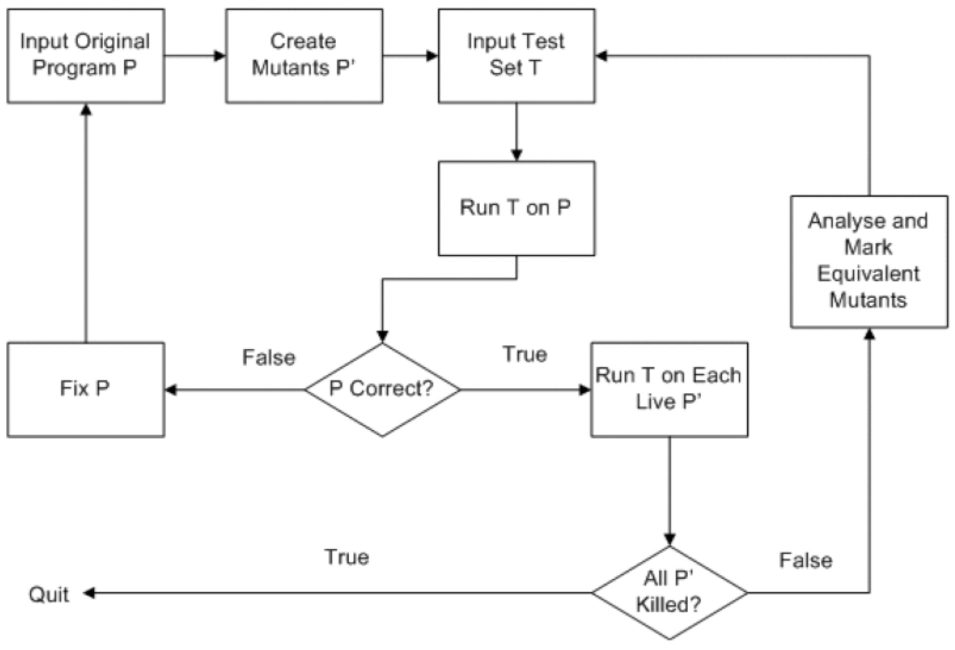

Figure 1: Generic process of mutation testing [2]

Mutation analysis involves generating a set of faulty programs, denoted as mutants p', from a given program p by making a few syntactic changes to the original code.

For instance, Table 1 shows the creation of mutant p' by changing logical operator, from && (and) to || (or).

Program p	Mutant p'
...	...
$if(a + b > 100$ && $c > 0)$	$if(a + b > 100 \mid\mid c > 0)$
$return$ 1;	$return$ 1;
...	...

Table 1: Example of mutant obtained by changing logical operator

Similarly, another example Table 2 shows that mutant p' is created by changing arithmetic operator, from $+$ (plus) to $-$ (minus).

Program p	Mutant p'
...	...
$if(a + b > 100$ && $c > 0)$	$if(a - b > 100$ && $c > 0)$
$return$ 1;	$return$ 1;
...	...

Table 2: Example of mutant obtained by changing arithmetic operator

Mutation operators are specific rules or transformations that introduce small changes (mutations) in the original code to create mutants. These typical mutation operators are specifically designed to alter variables and expressions through replacement, insertion, or deletion operations [2]. Among numerous mutation operator, Tables 1 and 2 are two examples of the mutation operator.

Offutt et al. [9] proposed the five-operator set as shown in Table 3. Defining mutant operators is relatively simple as it involves specifying syntactic changes. However, creating useful operators is generally a complex task. Past research has typically taken two approaches: initially defining a comprehensive set of mutant operators based on the language's grammar, and then through empirical studies, selecting specific subsets to enhance practicality and scalability. Both defining the operators and selecting representative subsets are interconnected, as they involve choosing from an extensive, potentially infinite pool of possibilities. Thus, forming small sets of operators can be seen as a subset selection from the vast range of potential options [1].

Names	Description
ABS	Absolute value insertion
AOR	Arithmetic operator replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
UOI	Unary operator insertion

Table 3: Offutt et al. [9] proposed five operator set [1]

In accordance with the selected set of mutant operators, a set of mutant instances are generated to conduct our analysis. Test objectives revolve around ”killing” these mutants, which entails designing test cases capable of detecting and eliminating all the mutants from the program under evaluation [1]. Once all the test cases have been executed, there might still be a small number of surviving mutants. To enhance the effectiveness of the test suite (denoted as T), the program tester has the option to introduce additional test inputs aimed at eliminating these surviving mutants.

Score

4 Challenges of Mutation Analysis

While Mutation Testing proves to be a valuable method for evaluating the adequacy of a test set, it encounters several challenges. Following are the major problems of mutation analysis [2]:

1. *Computationally Expensive*: Mutation Testing faces a significant hurdle due to its computational complexity. The process demands the execution of a large number of mutants against the test set, leading to considerable computational burden.

2. *Substantial Human Effort*: Mutation testing requires a substantial commitment of human effort. It involves intricate tasks that demand careful attention and meticulous execution from human testers.

   (a) *Human Oracle Problem*: The process of verifying the output of the original program with each test case is not unique to Mutation Testing only, but is present in all types of software testing. However, Mutation Testing is effective precisely because it is rigorous, leading to a higher number of test cases and, consequently, an increase in the cost of performing these validations (oracle cost). This cost is often the most expensive aspect of the entire testing process [10]. ** Paraphrase and modify content **

   (b) *Equivalent Mutant Problem*: It is important to note that there are certain mutants, known as ”Equivalent Mutants,” which cannot be eliminated since they always produce identical output to that of the original program. Although these mutants exhibit syntactic differences, they functionally behave the same as the original program. Due to the inherent challenge of determining whether two mutants are equivalent or not, identifying equivalent mutants typically requires additional human effort [1] [11].

While achieving an absolute resolution to these challenges may remain unattainable, noteworthy progress in Mutation Testing has paved the way for automation of the process and improved runtime, enabling a level of practical scalability as demonstrated in this survey. Prior research has predominantly concentrated on devising methodologies to mitigate computational expenses, a subject that will be explored in the subsequent discussion.

5 Application of Mutation Testing

Mutation testing is widely recognized for its flexibility, making it applicable across various testing domains. While primarily used in unit testing, advancements have extended its reach to specification, design, integration, and system levels. The method has been successfully applied to popular programming languages including C, C++, C# [12], Java, JavaScript, and Ruby, as well as specification and modeling languages. For instance, in the design, Mutation Testing has been employed in various contexts such as Finite State Machines [13]. These examples illustrate the diverse range of applications where Mutation Testing has been successfully utilized. Furthermore, it has been adapted for diverse programming paradigms such as object-oriented, functional, aspect-oriented, and declarative-oriented programming [1] [2].

6 Evaluation Metrices

One of the assessment of the test suite's effectiveness is quantified by the mutation score. A mutant that remains unaffected by any test case is considered an equivalent mutant, and mutation score is represented the ratio of killed mutants to the total number of non-equivalent mutants [12].

7 Implementation

By conducting an in-depth examination of the applications of mutation testing, specifically within programming contexts, we will proceed to implement mutation testing in a programming language of our choice, leveraging appropriate tools. This section will focus on documenting our implementation process, showcasing our understanding, and presenting the outcomes of our efforts.

8 Conclusion

In conclusion, mutation testing is a fault-based technique used to evaluate the quality of a test suite by introducing mutations into a program. While it offers valuable insights, challenges such as computational burden and human effort have prompted the development of cost reduction techniques. This study aims to deepen our understanding of mutation testing, implement it in a programming language, and explore its effectiveness in detecting errors in real-world software development.

References

[1] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," vol. 112 of *Advances in Computers*, pp. 275–378, Elsevier, 2019.

[2] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[3] P. Chevalley, “Applying mutation analysis for object-oriented programs using a reflective approach,” in *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pp. 267–270, 2001.

[4] A. T. Acree, T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward *Mutation analysis.*, 1979.

[5] L. Clarke, A. Podgurski, D. Richardson, and S. Zeil, “A formal evaluation of data flow path selection criteria,” *IEEE Transactions on Software Engineering*, vol. 15, no. 11, p. 1318–1332, 1989.

[6] W. E. Wong and A. P. Mathur, “Fault detection effectiveness of mutation and data flow testing,” *Software Quality Journal*, vol. 4, no. 1, p. 69–83, 1995.

[7] Z. Ahmed, E. Stein, S. Herbold, F. Trautsch, and J. Grabowski, “A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations,” May 2023.

[8] A. J. Offutt, “Investigations of the software testing coupling effect,” *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, p. 5–20, 1992.

[9] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, p. 99–118, 1996.

[10] J. S. Bradbury, J. R. Cordy, and J. Dingel, “Exman: A generic and customizable framework for experimental mutation analysis,” in *Second Workshop on Mutation Analysis (Mutation 2006 - ISSRE Workshops 2006)*, pp. 4–4, 2006.

[11] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, “A systematic review of the application and empirical investigation of search-based test case generation,” *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 742–762, 2010.

[12] A. Derezinska and K. Kowalski, “Object-oriented mutation applied in common intermediate language programs originated from c#,” in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pp. 342–350, 2011.

[13] A. Pretschner, T. Mouelhi, and Y. L. Traon, “Model-based tests for access control policies,” in *2008 1st International Conference on Software Testing, Verification, and Validation*, pp. 338–347, 2008.