

A Systematic Review on Mutation Testing Approach

Rajasekaran S¹, Sudhakar K²,

¹ Lincoln University College, Malaysia;

² Nitte Meenakshi Institute of Technology, Nitte (Deemed to be University), India;

Email ID: pdf.rajasekaran@lincoln.edu.my; ksudhakar.cs@gmail.com

Abstract: The main aim of software engineering is the development of projects that achieve the desired outcome within a budget and a specified timeframe. Recently many mutation testing tools have been proposed that rely on bug-fix patterns and natural language models trained on large code corpus. As these tools operate fundamentally differently from the grammar-based traditional approaches, a question arises of how these tools compare in terms of fault detection and cost-effectiveness. Simultaneously, mutation testing research proposes mutant selection approaches based on machine learning to mitigate its application cost. This raises another question: How do the existing mutation testing tools compare when guided by mutant selection approaches? To answer these questions, we compare four existing tools μ BERT (uses pre-trained language model for fault seeding), IBIR (relies on inverted fix-patterns), DeepMutation (generates mutants by employing Neural Machine Translation) and PIT (applies standard grammar-based rules) in terms of fault detection capability and cost-effectiveness, in conjunction with standard and deep learning based mutant selection strategies.

Keywords: Software Testing; Budget; Mutation testing; Software Testing methods; Testing Tool performance.

1. Introduction

Mutation testing is a quality-assurance technique that injects small, artificial errors into program code to evaluate how well an existing test suite catches faults. Mutation technique in software testing is considered as the most fascinating way to validate the software under analysis. It is the process whereby a fault is deliberately inserted into a software system, in order to assess the quality of test data, in terms of its ability to find this fault. Mutation testing is also used as a way to drive the test data development process. Traditionally, faults were inserted one by one into a software system, but more recently there has been an upsurge of interest by the area of higher-order mutation, in which multiple faults are inserted into the system at once. [1].

Powerful learning-based mutant selection strategies have been proposed recently [2-4], intending to reduce the application cost and noise of mutation testing, which has been for long considered as a primary cause that keeps the technique away from broad industrial service. These strategies aim at discarding redundant mutants and providing testers with mutants that will bring value to the testing process. Typically, these strategies employ user-defined features- code features learned using deep learning- independent of how mutants are introduced. Since these mutant selection strategies give different importance to mutants, this may affect the cost effectiveness of mutation testing and raises the question of how the different tools compare in terms of fault detection under mutant selection strategy guidance.

2. Related work

An ST (Systematic Testing) is a process to identify the differences between current condition of software and desired condition. The aim of ST is to develop the software through which determine whether the software achieves its requirements, and to ensure that the software does not contain

any errors. In fact, most of the resources of a software project (more than 50%) are invested in the detection and correction of faults [1-2]. Despite this massive investment, it is widely known that the complexity of software makes it impossible to detect all the faults of software [1].

ST depends on generating test cases which are collection of test actions that are executed to verify a specific attribute or function of the software application. There are many testing techniques used to design the test cases. Some of these techniques work on designing highly effective test cases that detects more errors with less effort and time [6-7]. These techniques can be categorized according to source code knowledge such as black-box, white-box, gray-box techniques [6], used information source knowledge such as program-based, specification-based, interface-based techniques [7-9], or testing level knowledge such as unit-level, integration-level, system-level techniques [10-11].

Mutation Testing is a fault-based software testing technique that has been widely studied for over three decades. The literature on Mutation Testing has contributed a set of approaches, tools, developments and empirical results which have not been surveyed in detail until now. This paper provides a comprehensive analysis and survey of Mutation Testing. The results of several development trend analysis, provides evidences that Mutation Testing techniques and tools are reaching a state of maturity and applicability, while the topic of Mutation Testing itself is the subject of increasing interest. This Systematic Literature Review aims to present the research carried out in the field of search-based approaches applied particularly to mutation testing.

Table 1. Compares this work with the related work or previous research by other researchers

Review	Strength	Limitations
Competent Programmer Hypothesis (CPH)	Tend to develop programs close to the correct version.	Only Simple faults can be corrected by a few small syntactical changes.
Coupling Effect	Complex faults can be corrected	High computational time and costs.
PI Test (PIT)	Applies standard grammar-based rules	Limited custom mutation operators
Bidirectional Encoder Representations from Transformers (μBERT)	Uses pre-trained language model for fault seeding.	High computational time and costs
Information-retrieval / Bug-report-driven nverse Repair framework (IBIR)	Relies on inverted fix-patterns	Limited diversity of fault patterns

Table 2. Features of existing Mutations Testing Tools

Review	Focus	Features
DeepMutation	Feed-Forward Networks (FNNs)	Evaluates test data quality using 8 source-level and 8 model-level mutation operators.
DeepMutation++	Recurrent Neural Networks (RNNs)	Extends DeepMutation to stateful structures like LSTMs/GRUs; supports static and dynamic runtime mutation.
DeepCrime	Real-world DL Faults	Mutates code <i>before</i> training begins using 24 operators derived from empirical studies of real bugs.
This work	Approach for Systematic Fault Analysis.	Designed for Deep Learning based Software Systems

3. Key Contributions

Although the existing Mutation Testing is able to effectively assess the quality of a test set, it still suffers from a number of problems. One problem that prevents Mutation Testing from becoming a practical testing technique is the high computational cost of executing the enormous number of mutants against a test set. The other problems are related to the amount of human effort involved in using Mutation Testing for handling equivalent mutants. Another challenges are adopting complex industrial systems and handling AI generated code.

Table 3. Statistics of Existing Mutation Testing Tools

Stage	Overall Tools	Academic Tools	Open Source Tools	Commercial / Industrial Tools
1975-1999	8	7	0	1
2000-2009	28	19	7	2
2010 - present	74	21	45	8

Table 4. Research Questions and Research Gaps

Research Questions	Research gaps
Can the existing systems cost optimized?	High Computational Cost and Retraining Overhead
Can the existing systems handle equivalent mutants efficiently?	Issue In handling equivalent mutants.
Is the existing "Kill" Criteria and Mutation Score Metrics appropriate for DL systems?	Flawed Test "Kill" Criteria and Mutation Score Metrics.
Does the existing tool supports Modern AI Paradigms?	Poor Integration with Diverse and Modern AI Paradigms.
Do we have Mutation testing tool especially for Deep learning based software systems?	Lack of Mutation testing tool especially for Deep learning based software systems.

4. Method, Experiments and Results

To address the research gaps, here is a necessity to develop a Novel Mutation Testing Approach for Systematic Fault Analysis especially in Deep Learning-Based Software Systems and the process of *MT* is accomplished in three steps as follow:

- First, inserting faults into the original program by making simple syntactic changes to produce a set of faulty programs called mutants; each mutant contains a different syntactic change.
- Second, applying set of test cases on the original program and also on all mutants. Then the ability of these test cases on detecting errors are assessed by calculating the mutation score (MS). MS measures the effectiveness of a test set in terms of its ability to detect faults.
- Third, comparing the results of the original program and the mutated one. If the result of executing a mutant differs from the result of executing the original program for any test cases in the input test set, the fault denoted by the mutant is killed or detected; otherwise it is said to be survived.

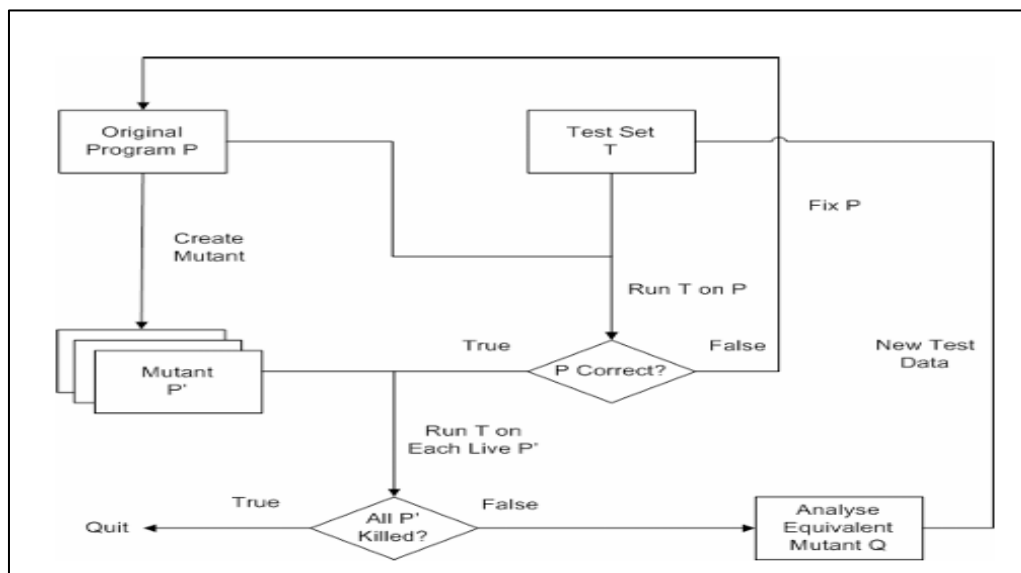


Figure 1. Genetic Process of Mutation Analysis

As per the results produced by Milos Ojdanic et al. [9] We can understand the effectiveness of PIT Default (58%), μ BERT (57%) and IBIR (54%) is similar, outperforming PIT (50%) when analyzing as many mutants as PIT Default. When writing the same number of tests, we observe that μ BERT reaches similar effectiveness (54%) as PIT Default, PIT 53% and IBIR 56%. When we focus on μ BERT, we can observe that both μ BERT (73.2%) and IBIR (72.7%) are more effective than PIT (65.7%) under the same effort. These differences also have statistically significant p-value, and $\hat{A}_{12}$ when compared to their cost-effectiveness, evidencing that μ BERT and IBIR in $\approx 99\%$ cases can detect more faults. Moreover, to reach the same effectiveness as PIT, μ BERT and IBIR need to analyze 44% and 38% fewer mutants than PIT. When we compare the number of tests, we observe that IBIR (65.2%) and μ BERT (64.01%) can detect near 4% more faults than PIT (61.16%) under the same effort. Finally, we can observe that IBIR (near 90%) is more effective than PIT (74%) when the same number of mutants are analyzed. IBIR needs to analyze 80% fewer mutants (and 25% fewer tests) than PIT to reach the same effectiveness. This difference is also statistically significant p-value <0.01 . This previous results shows the minimum performance of the existing Mutation Testing tools and the necessity of new proposed tool.

5. Conclusion

We studied the fault detection performance of recently proposed mutation testing tools (DeepMutation, PIT, IBIR and μ BERT) on a new and large fault dataset. We also understand the two different mutant selection strategies such as standard and state-of-the-art deep learning driven one, then they performed a cost-effectiveness comparison using two typically adopted cost models; one associated with the number of mutants requiring analysis and a second one with the number of tests to reveal the injected faults. Our results showed that IBIR has the highest fault detection capability ($\approx 90\%$ on average) but is not the most cost-efficient. In contrast, μ BERT, even less effective, has significantly higher cost-effectiveness when using learning-based selection strategies, approximately 12% higher, than all the other tools. More notably, we found that mutation testing tools perform differently when guided by mutant selection strategies, indicating the need for considering intelligent selection when comparing mutation testing tools. In other words, mutant selection strategies must be considered when comparing the cost-effectiveness of mutation testing tools to avoid the risk of making wrong conclusions. Still, there is a need for developing a novel Mutation testing tool for systematic fault analysis especially for Deep learning-based Software Systems.

References

- [1] JiaY. et al. Higher order mutation testing J. Inf. Softw. Technol. , 2009.
- [2] LangdonW.B. et al. Efficient multi-objective higher order mutation testing with genetic programming J. Syst. Softw. , 2010.
- [3] OmarE. et al. Subtle higher order mutants Inf. Softw. Technol. , 2017.
- [4] BezierB. Software Testing Techniques , 1990.
- [5] MyersG.J. et al. The Art of Software Testing , 2004.
- [6] CopelandL. A Practitioner's Guide To Software Test Design, 2003.
- [7] BinderR.V. Testing Object-Oriented Systems: Models, Patterns, and Tools ,1999.
- [8] KapfhammerG. The Computer Science Handbook, Chapter Software Testing , 2004.
- [9] Milos Ojdanic et al. On Comparing Mutation Testing Tools through Learning-based Mutant Selection, IEEE/ACM International Conference on Automation of Software Test (AST) ,2023.
- [10] Samuel J Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and Ren'e Just. Prioritizing mutants to guide mutation testing. In Proceedings of the 44th International Conference on Software Engineering, pages 1743–1754, 2022.
- [11] Renzo Degiovanni and Mike Papadakis. μ BERT: Mutation testing using pre-trained language models. In Mutation Workshop at ICST. IEEE, 2022.
- [12] Milos Ojdanic, Ezekiel Soremekun, Renzo Degiovanni, Mike Papadakis, and Yves Le Traon. Mutation testing in evolving systems: Studying the relevance of mutants to code evolution. ACM Transactions on Software Engineering and Methodology, 2022.
- [13] Cedric Richter and Heike Wehrheim. Learning realistic mutations: Bug creation for neural bug detectors. In 2022 IEEE Conference on Software Testing, Verification and Validation (ICST), pages 162–173, 2022.
- [14] Milo's Ojdanić, Wei Ma, Thomas Laurent, Thierry Titchou Chekam, Anthony Ventresque, and Mike Papadakis. On the use of commit relevant mutants. Empirical Software Engineering, 27(5):1–31, 2022.