ISSN 2347 - 3983

Volume 8. No. 4, April 2020

International Journal of Emerging Trends in Engineering Research

Available Online at http://www.warse.org/IJETER/static/pdf/file/ijeter25842020.pdf

https://doi.org/10.30534/ijeter/2020/25842020

Comparative Analysis between Selective and Non-Selective Mutation Techniques

Samah W.G. AbuSalim[1], Rosziati Ibrahim[2], Jahari Abdul Wahab[3]

[1,2] Department of Software Engineering, University Tun Hussein Onn Malaysia,Parit Raja, Johor, 86400, Malaysia

samahwasalim@gmail.com, rosziati@uthm.edu.my

[3]Engineering R&D Department, Sena Traffic Systems Sdn. Bhd., Kuala Lumpur, 57000, Malaysia

jahari@senatraffic.com.my

ABSTRACT

Software testing is the process to provide users with information about the quality of the product or software being tested. One of most effective testing techniques is Mutation testing. Mutation testing is a software testing technique that injects a syntactic alteration into the source code to construct mutants. This technique works by testing a test suite's ability to detect mutants that are used to determine a test suite's performance. The information from mutation testing can be used to improve the quality of a test suite by developing additional test cases. Despite its effectiveness, many factors make it costly and therefore difficult to use by software testers for example, execute large sets of mutants, and generate many numbers of test cases to kill the mutants and equivalent mutants. Several approaches have been proposed in order to reduce the mutation cost by reducing the number of mutants generated which will lead to a reduced execution time. Thus, this study aims to employ an appropriate mutant reduction technique by comparing and analyzing between two approaches namely selective and non-selective mutants. To achieve this, mutation operators were generated for each approach. Test cases were then generated by using JUnit testing framework. Performance measurements in terms of mutation coverage and execution time for each case study were evaluated by using PITest tool, which is mutation testing tool for Java. Finally, the mutation coverage and execution time were compared and analyzed among the four case studies used namely Cal, Airline Reservation, Chess and Elevator applications. The experimental results showed that selective mutant operators can drastically reduce the execution time and make mutation testing more e□ective since the execution time for selective mutant is 8.5 seconds while non-selective mutant takes 27.25 seconds in execution time.

Key words: Mutation Cost Reduction Techniques; Mutant Operators; Mutant Reduction Techniques; Mutation Testing.

1. INTRODUCTION

Software testing is one of the important and primary parts of software quality assurance (SQA). Through software testing, software faults are shown to users, but bugs and errors often prevail, even after programmers frequently test the entire site and its functions [1]. The objective of testing is not only to capture bugs and system defects; it can be performed for other reasons such as software quality assurance, software verification and validation and software functionality checking [2]. Software testing can be performed automatically. Automation is used in various control systems for running machinery and other operations with the least human interference [3]. In the field of software testing, mutation testing which first proposed by DeMillo et al. [4]. Mutation testing is one of the most effective techniques for quality evaluations of input values and test cases. While mutation testing is effective, it is still considered as an expensive and time-consuming complicated technique that has prompted many researchers to find ways to reduce and minimize costs. The implicit running cost of performing the large number of mutants against the test set is one of the reasons why mutation testing is so expensive [5]. Therefore, a lot of research has discussed the problem of how to reduce the number of mutants produced without affecting the efficiency of the test. For this reason, four techniques are introduced to reduce the number of mutants, Mutant Sampling, Mutant Clustering, Selective Mutation and Higher Order Mutation [6]. In this paper, a comparative study between two mutant reduction techniques that are mutant selection technique and non-selection technique (All-Operators) for Java apps using PITest mutation testing tool [7] are analyzed to conduct the best mutant reduction technique. This paper consists of 7 Sections. Section 2 looks into a different view of previous researchers and their conclusions through an overview of the literature relating to mutant reduction techniques and methods. Several mutant reduction techniques are discussed in Section 3. Section 4 presents the methodology of this study which include three phases. Section 5 shows the results of the empirical study. Section 6 proposes the comparative analysis between two mutant reduction techniques based on number of mutant generated, execution time and mutation score and the concluding remarks are in Section 7.

2. RELATED WORK

Falah et al. [8] present a strategy named Random Selective Mutation (RSM) that reduces the cost of testing during the mutation creation phase by reducing the number of mutation operators used at this phase. A hypothesis was relied on in their technique, that is the production of fewer mutants is sufficient to conduct highly efficient tests based on the production of fewer mutations. Their experiment showed that mutation testing can be achieved with a small subset of mutation operators. It has suggested that choosing the type of mutation operator can affect the efficacy of mutant detection and they confirmed that method operators are general used in smaller size applications.

Hamimoune and Falah [9], introduce a comparative analysis between four mutation testing techniques (class-level operators, method-level operators, all operators, and random sampling) using MuClipse mutation too for Java language in order to find the most effective technique for mutation testing. They suggested that the mutant detection effectiveness is based on the selecting of the type of mutation operators. They also show that for small and medium-size application, the most efficient and prevalent operator is all sampling operator.

Kurtz et al. [10] analyze selective mutations without the noise created by the inherent replication of traditional mutations. Then, evaluated comprehensively small groups of mutation operators for the Proteum mutation system, which is a C mutation tool and assess the dominator mutation scores and the work required for each. The results show that selective methods need to become more sophisticated in order to achieve high performance for mutation process.

Delgado-Pérez & Bulo [11] use the Evolutionary Mutation Testing (EMT) technique which allows the number of mutants to be reduced while retaining the power to refine the test suite. Their experiments applied with six real-world C++ case studies using the tool GiGAn with MuCPP mutation tool for C++. The results indicate that EMT work well for most case studies and test suite improvement demand rates.

Chekam et al., [12] implement approach known as FaRM to machine learning, which learns to select fault revealing mutants using collection of tools in C++. Their results show that FaRM performs well than all existing methods of mutant selection.

Gopinath et al., [13] compared random sampling with multiple reduction techniques in order to find which technique is most effective in cost reduction. The experiment applied in real-world open source programs using PIT tool. They found that none of mutation reduction techniques that evaluated produced an efficacy advantage of more than 5% compared to random sampling. Their research shows that blind random sampling of mutants performs better and more effectiveness than current mutation reduction approaches.

Rani and Chaudhary [14] define an improved clustering approach to improve the mutation testing. Their work is characterized in two main phases. In first stage, the code segment extraction in done. In second stage, the cost analysis and clustering approaches are applied to improve the mutation process. The experimentation is performed on multiple code blocks using MuJava tool. The results show that the mutation testing can be enhanced by reducing the killed mutants over the code.

Abuljadayel and Wedyan [15] introduce approach using genetic algorithm to create difficult higher order mutants to kill and reduce the equivalent mutants produced. They developed a Java tool named HOMJava to perform their approach. Their study shows that the mutants that generated by their approach were difficult to kill and the remained equivalent mutants were about 4% from 100 sample of possible equivalent mutants and can killed them by simple test case.

Table 1 summarize the literature review in mutant reduction techniques which are: Selective Mutant, Mutant Sampling, Clustering Mutant and Higher Order Mutant.

Table 1: Summary of comparative studies in related works

Author and year	Techniques	Framework/Method
Falah et al., (2015) [8]	Selective Mutant	The results showed that mutation testing can be done with a small subset of mutation operators.
Hamimoune & Falah (2016) [9]	Selective Mutant	They suggested that the mutant detection effectiveness is based on the selecting of the type of mutation operators.
Kurtz et al., (2016) [10]	Selective Mutant	They suggest that selective methods will have to become more advanced in order to achieve high efficiency for complete mutation analysis.
Chekam et al., (2018) [12]	Selective Mutant	Their results show that their approach outperforms all existing methods of mutant selection.
Gopinath et al. (2017) [13]	Mutant Sampling	Their result shows that blind random sampling more efficacy than current strategies for mutation reduction.
Rani and Chaudhary (2015) [14]	Mutant Clustering	The results show that their work in reducing the killed mutants over the code has significantly enhanced the mutation testing.
Abuljadayel & Wedyan (2018) [15]	High Order Mutant	Their strategy can produce hard higher order mutants to kill and minimize the number of equivalent mutants.

Based on Table 1, there are many researches proposed by several people on the field of mutation testing. Many researchers tried to reduce the cost of mutations by reducing the number of mutations produced in a phase of mutation analysis. However, there are a small number of researches done comparative analysis between selected mutant operators and all mutant operators using PIT mutation testing tool. Therefore, this circumstance has motivated this study to compare between two mutant reduction techniques selective mutant and non-selective (all-operator) mutant using PIT tool.

3. MUTANT REDUCTION TECHNIQUES

Mutant reduction techniques have been classified as Mutant Sampling, Mutant Clustering, Selective Mutation and Higher Order Mutation. This section will show these mutant reduction techniques briefly.

3.1 Mutant Sampling

It is first suggested by Acree [16] and Budd [17]. It is a technique that randomly selects a small group of mutants. Similarly, to traditional process of mutation testing, first all potential mutants are generated. Then X% are selected randomly from these mutants for mutation analysis and discarded residual mutants. The main focus was on random selection rate choice (x) [18]. This method has been the topic of many empirical studies. In [19] the authors tested mutant sampling in Java programs at random. The results showed that random sampling of mutants by 60% or 50% can reduce the cost of testing in Java program with adequate score for mutations and code coverage. In [18] the authors claim that the best results can be achieved in a mutation testing when 90% samples are taken from all mutants that the test is being performed.

3.2 Selective Mutant

This technique was proposed first, as “constrained mutation” by Mathur [20] and Offutt et al. [21], later this concept extended by naming it Selective Mutation. It is not random as mutation sampling, but instead aims to reduce the number of mutants by using a subset of the available mutation operators without gaining a significant loss of test e□ectiveness relative to non-selective mutant (all operators). Wong [22] researched the concept of using a uniform distribution to pick mutants randomly. He confirmed, that the findings were poor when use of mutants selected at random. The appearance of this negative result was the reason for the emergence of a selective mutation technique by Wong and Mathur [23].

3.3 Mutant Clustering

In Hussain's master thesis, the concept of Mutant Clustering was first suggested [24]. Instead of randomly picking mutants, clustering algorithm was used to select a group of mutants. The Mutation Clustering process begins with the generation of all mutants of first order. A clustering algorithm is then implemented on the basis of the killable test cases to identify the first order mutants into separate clusters. Same set of test cases are expected to destroy every mutant in the same cluster. Same test cases is expected to destroy every mutant in the same cluster. Only a small number of mutants are chosen for use in mutation testing from each cluster, the remaining mutants are discarded [6].

3.4 Higher Order Mutant

This approach is performed by injecting two or more faults into the mutated program. The number of injected faults is representing the order of mutant. For example, the mutant of second order means that mutant has two faults while mutant of third order means that mutant has three faults [15]. To produce Higher Order Mutant (HOM), the mutation operators are performed several times, which typically varies from the original program being evaluated by simple changing [25]. Several studies indicated that HOMT can help identify equivalent mutants and improve effectiveness.

4. RESEARCH METHODOLOGY

This section discusses the methodology of this study which include three phases, as shown in Figure 1.

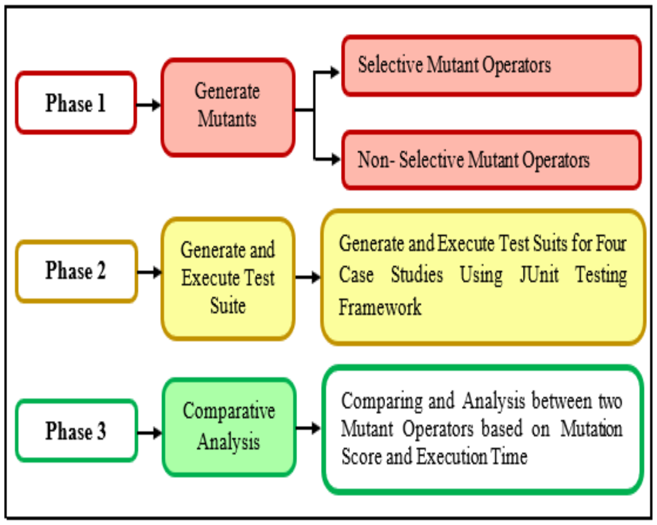
Phase 1
Generate Mutants
Selective Mutant Operators
Non- Selective Mutant Operators
Phase 2
Generate and Execute Test Suite
Generate and Execute Test Suits for Four Case Studies Using JUnit Testing Framework
Phase 3
Comparative Analysis
Comparing and Analysis between two Mutant Operators based on Mutation Score and Execution Time

Figure 1: Research Phases

The explanation of these phases is discussed in the following sections.

4.1 Phase 1 (Generating Mutants based on Mutant Reduction Technique) : The process of mutation normally consists of three steps: Generating mutants, executing the test suite, and analyze the outcome. After the final step, the tester can use the information given from the analysis to design additional test suits to strengthen the test suite and improve the mutation score.

4.1.1 The Process of Mutation Testing

As illustrated in Figure 2, the first step of mutation analysis is to generate mutants of the original program. This is done by inducing small syntactic changes according to well-formed rules called mutation operators. Mutants are created by applying an operator on the source code. PIT mutation testing tool [28] include two type of mutant operators. The first type of mutant operators is activated by default while the second type is deactivated by default as shown in [7]. For selective mutant, some operators like operators that activated by default was selected while non-selective means all operators activated and deactivated by default were applied; thus, every mutant will contain a single syntactic change that differs from the original program.

After generated mutants, a test cases are carried out against the original program and the mutants to compare the output from both original and mutants' programs. If the output di□ers from the original program, then the mutant is said to be detected or killed, otherwise it is said to have survived [6]. If any mutants remained undetected, the surviving mutants can by killed through mutant analysis, which can help developing additional test cases. Mutation testing tools normally provide some sort of information about the mutants that can help the augmentation of the test suite. After test case execution the surviving mutants need to be analyzed. This is either done to mark equivalent mutants since they need to be omitted from the final mutation score, or to use the information to develop additional test cases. The end result will produce a mutation score. The mutation analysis aims to increase the mutation score close to 1, i.e. 100% [27]. Mutation Score is defined as the ratio of mutants that are killed by our test cases to total number of mutants. Mutation score can calculate by the following formula:

Mutation Score = (Killed Mutants / Total number of Mutants) * 100 (1)

Where

0 <= M.S. <= 0.9

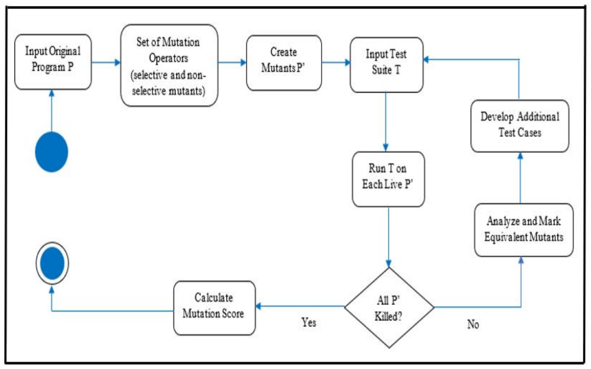
Input Original Program P
Set of Mutation Operators (selective and non-selective mutants)
Create Mutants P'
Input Test Suite T
Develop Additional Test Cases
Run T on Each Live P'
Analyze and Mark Equivalent Mutants
All P' Killed?
Yes
No
Calculate Mutation Score

Figure 2: The process of mutation analysis

4.1.2 Mutant Classification

The mutants can be classified into 3 different categories. The first one called Redundant mutants [10] which can be killed by every test case, thus not adding any useful test cases to the test suite, and omitting operators producing high numbers of redundant mutants would reduce the execution time. As compared to Redundant mutants, there are hard to kill mutants [28]. Such mutants are particularly interesting as they contribute to strong test cases as they are only killed by a small percentage of test cases. The idea is that redundant mutants subsume hard-to-kill mutants, so one can exclude operators that tend to produce redundant mutants without a significant loss in the test case. some mutants do not modify the meaning of the original program, means that mutant program equivalent to the original program, and thus are not a fault. These mutants are considered equivalent mutants. For example, a mutation operator might change the condition in a for-loop, as shown in Figure 3.

Program p	Equivalent Mutant m
for $(int\ i = 0;\ i < 10;\ i{+}{+})$ { ...$(the\ value\ of\ i$ $is\ not\ changed)$ }	for $(int\ i = 0;\ i\ != 10; i{+}{+})$ { ...$(the\ value\ of\ i$ $is\ not\ changed)$ }

Figure 3: Example of Equivalent Mutation [6]

In the above example, the code has changed, but the for-loop behavior is the same, which makes the mutant equivalent since no test case can detect it. A mutant might also be unkillable in a programming language because of constraints or limitations. Even for small programs, the effort to detect equivalent mutants can be strong, causing unnecessary device processing and the software engineer have to spend energy on anon-existent problem.

4.2 Phase 2 (Generating and Executing Test Suites Using JUnit Testing Framework): The generated of test suites for four case studies have created by hand in the JUnit testing framework, and additional test cases were generated by analyzing the surviving mutants. Only test cases that were responsible for killing at least one mutant were added into the final test suite. The process of test case generation continued until all non-equivalent mutants were killed and all equivalent mutants have been marked. When all the necessary test cases, for each mutant operator had been created, each test suite were executed against the original program and the mutants to compare the output from the test suite and a mutation score was calculated. The test suites have 4 columns: test suite id (TS_ID), test suite name (TS_Name), test case id (TC_ID) and test case description), as shown in Table 2.

Table 2: Test Suite Table

TS_ID	**TS_Name**	**TC_ID**	**Test Case Description**
TS_1	Flights	TC_01	To verify flight details like flight name, code, from and to destination and timing.

Table 3 describes the four case studies that used to do this research. The Java applications have been downloaded from GitHub website.

Table 3: Programs used in the experiments

Java programs	Lines of Code	Methods	Description
Cal	14	1	Calculate the number of days in the same year, between the two days.
Airline Reservation	154	41	Provide online ticket and seat booking for national and international flights as well as flight departure information.
Chess game	352	62	Game between two users. The users will enter their moves by putting in the column and row from which they want to move and then the location they want to move to.
Elevator	434	82	A small Java project to simulate the evolution of an elevator.

4.3 Phase 3 (Comparing and Analysis between two Mutant Reduction Techniques): The last phase of our methodology is do comparative analysis between the selective and non-selective mutant. The results are compared and analyzed after they are obtained by execution of the test suite by using PIT mutation testing tool based on three measurements which are Number of mutants generated, Mutation Score and Execution Time of four case studies.

5. RESEARCH RESULTS

The results for both mutant reduction techniques are compared and analyzed for four case studies. after they are obtained by execution of the test suits. Three measurements are used: Number of mutants generated, Mutation Score and Execution Time

5.1 Results of Number of Mutants Generated

In this section, the generating of the mutants using PIT tool for two mutant reduction techniques: Selective Mutation and Non-Selective (All-Operators) Mutation for the four case studies is presented. The mutation operators are shown in Table 4.

Table 4: Results of Number of Mutants Generated

Application	Selective Mutant	Non-Selective (All-Operators) Mutant
Cal	17	60
Airline Reservation	72	264
Chess	68	64
Elevator	291	805

5.2 Results of Mutation Score

The mutation score for the mutation testing is represents in per cent. Thus, the mutation score for Selective mutation and Non-Selective (All-Operator) mutation for four cases studies is represented in Table 5 and Table 6.

Table 5: Results of Mutation Score for Selective Mutants

Applications	Class		Unit Testing			
	Name	# of methods	Killed mutants	Total Mutants	Score (%)	Mean Score
Cal	Cal	1	10	17	59	59
Airline Reservation	Main	1	0	21	0	50
	Database	11	18	33	55	
	Flight	9	7	7	100	
	Passenger	5	3	3	100	
	Seat	5	3	3	100	
	Ticket	10	5	5	100	
Chess	Bishop	3	4	4	100	68
	Board	10	20	43	47	
	Game	19	31	69	45	
	King	3	13	14	93	
	Knight	3	14	14	100	
	Main	1	0	1	0	
	Pawn	3	20	20	100	
	Piece	14	56	72	78	
	Queen	3	7	7	100	

	Rook	3	4	4	100	
Elevator	Building	9	15	28	54	41
	Doors	5	3	13	23	
	Elevator	17	17	72	24	
	FIFOElev	8	21	22	95	
	Floor	7	11	35	31	
	Naive Elev	8	18	34	53	
	Person	12	28	48	58	
	Person Exception	2	0	2	0	
	Simulation	3	0	13	0	
	Timer	5	5	8	63	
	Main	5	0	16	0	

Table 6: Results of Mutation Score for Non-Selective (all-operators) Mutants

Application	Class		Unit Testing			
	Name	# of methods	Killed mutants	Total Mutants	Score (%)	Mean Score
Cal	Cal	1	22	60	37	37
Airline Reservation	Main	1	0	67	0	45
	Database	11	44	119	37	
	Flight	9	24	24	100	
	Passenger	5	9	9	100	
	Seat	5	9	10	90	
	Ticket	10	34	35	97	
Chess	Bishop	3	10	10	100	64
	Board	10	48	96	50	
	Game	19	108	254	43	
	King	3	34	38	89	
	Knight	3	44	46	96	
	Main	1	0	2	0	
	Pawn	3	57	63	90	
	Piece	14	141	193	73	
	Queen	3	20	20	100	
	Rook	3	10	10	100	
Elevator	Building	9	41	70	59	45
	Doors	5	16	39	41	
	Elevator	17	48	150	32	
	FIFO Elev	8	69	74	93	
	Floor	7	33	91	36	
	Naive Elev	8	58	128	45	
	Person	12	88	153	58	
	Person Exception	2	0	3	0	
	Simulation	3	0	39	0	
	Timer	5	10	17	59	
	Main	5	0	40	0	

5.3 Results of Execution Time

The execution time is very important element in any system. Table 7 shows that Selective and Non-Selective mutation execution time.

Table 7: Average of Execution Time in Seconds

Program	Selective Mutant (s)	Non-Selective Mutant (s)
Cal	2	7
Airline Reservation	5	8
Chess	16	47
Elevator	11	47

For the four case studies, the formal specification can also be formed using Z specification for example. Some researchers used formal notation prior to implementation such as [29], [30] and [31].

6. COMPARATIVE ANALYSIS

The experimental results that have been acquired from the implementation of each technique separately on mutation testing can be compared in terms of:

6.1 Comparison of Number of Mutants Generated

The number of mutants reflects upon the execution time, and therefore plays an important role when assessing the e□ciency of operators. As shown in Figure 4, the Non-Selective mutant, produces on average the highest number of mutants for all applications. Non-selective (all-operators) mutant is better than selective mutant in term of number of mutants generated.

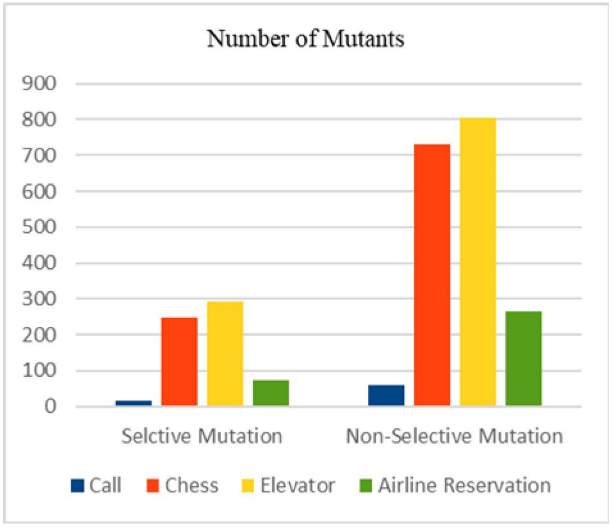

Figure 4: The number of mutants each set of operators produced for every program

6.2 Comparaison of Mutation Score

The mutation score that each test suite, adequate for a specific set of mutation operators, managed to produce selective mutation and non-selective mutation operators (all operators). As shown in Figure 5, the selective mutation, coverage the highest number of mutants. So, selective mutant is better than non-selective (all-operators) mutant in term of mutation score.

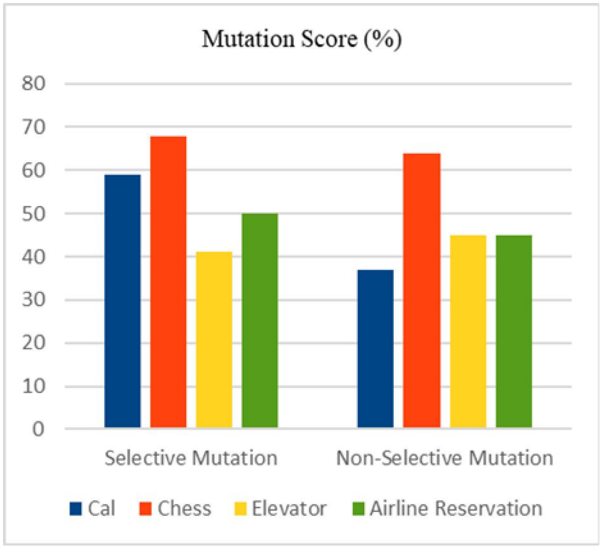

Figure 5: The mutation score of techniques produced for every program

6.3 Comparison of Execution Time

The execution time is very important element in any system. Table 8 shows that Selective mutation obtained the shortest time. Selective mutant is better than non-selective (all-operators) mutant in term of time execution.

Table 8: Average of Execution Time in Seconds

Program	Selective Mutant (s)	Non-Selective Mutant (s)
Cal	2	7
Airline Reservation	5	8
Chess	16	47
Elevator	11	47
Average (s)	8.5	27.25

7. CONCLUSION

In this research, two mutant reduction techniques were compared and analyzed in terms of number of mutators generated, mutation coverage and execution time. To achieve this, mutation operators were generated for each approach. Test cases were then generated by using JUnit testing framework. Performance measurements in terms of mutation coverage and execution time for each case study were evaluated by using PIT tool. Finally, the mutation coverage and execution time were compared and analyzed among the four case studies namely Cal, Airline Reservation, Chess and Elevator applications. The experimental results showed that selective mutant operators can drastically reduce the execution time and make mutation testing more e□ective. As the execution time for selective mutant is 8.5 seconds while non-selective mutant takes 27.25 seconds in execution time. For the future, there is a need to conduct more research in this particular field of study including more techniques of cost reduction of mutation testing like Mutant Sampling and Clustering mutation. Furthermore, many studies can also be applied to compare between mutation testing techniques with other techniques like regression testing technique and white-box testing technique.

ACKNOWLEDGMENT

This project is funded by the Ministry of Education Malaysia under the Malaysian Technical University Network (MTUN) grant scheme Vote K234 and SENA Traffic Systems Sdn. Bhd.

REFERENCES

[1] A. Thioac, E. J. Domingo, R. M. Reyes, N. Arago, R. Jr. Jorda, J. Velasco. **Development of a Secure and Private Electronic Procurement System based on Blockchain Implementation**, International Journal of Advanced Trends in Computer Science and Engineering, Volume 8, No. 5, September - October 2019, Available Online at

http://www.warse.org/IJATCSE/static/pdf/file/ijatcse115852019.pdf

[2] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L.Traon, & M. Harman. **Mutation Testing Advances: An Analysis and Survey**, Advances in Computers, 2018. https://doi.org/10.1016/bs.adcom.2018.03.015

[3] H. Kumar, S. Kamboj. **Modelling of Automatic Material Handling System using PLC**, International Journal of Advanced Trends in Computer Science and Engineering, Volume 8, No. 3, May - June 2019, Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse90832019.pdf

[4] R. A. DeMillo, R. J. Lipton, & F. G. Sayward. **Hints on Test Data Selection: Help for the Practicing Programmer,** IEEE Computer, 11(4), 34–41, Apr.1978. https://doi.org/10.1109/C-M.1978.218136

[5] P.K. Chaurasia. **Mutation Testing: A Review**, Journal of Global Research in Computer Science. Volume 5, No. 2, February 2014.

[6] Y. Jia and M. Harman. **An Analysis and Survey of the Development of Mutation Testing,** IEEE Transactions on Software Engineering, 2011.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis, & A. Ventresque. **PIT: a practical mutation testing tool for Java (demo)**. Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016. doi:10.1145/2931037.2948707

[8] B. Falah, M. Akour & S. Bouriat. **RSM: Reducing Mutation Testing Cost Using Random Selective Mutation Technique,** Malaysian Journal of Computer Science. 28. 338-347. 10.22452/mjcs. vol 28 no 4 (2015). https://doi.org/10.22452/mjcs.vol28no4.5

[9] S. Hamimoune and B. Falah. **Mutation testing techniques: A comparative study,** 2016 International Conference on Engineering & MIS (ICEMIS).

[10] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, & N. Gökçe. **Analyzing the validity of selective mutation with dominator mutants,** Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016. https://doi.org/10.1145/2950290.2950322

[11] P. Delgado-Pérez and I. Medina-Bulo. **Search-Based Mutant Selection for Efficient Test Suite Improvement: Evaluation and Results,** Information and Software Technology. (2018).

[12] T.T. Chekam, M. Papadakis, T. Bissyandé, Y.L. Traon, & K. Sen. **Selecting Fault Revealing Mutants,** Empirical Software Engineering. (2018).

[13] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, & A. Groce. **Mutation Reduction Strategies Considered Harmful,** 2017 IEEE Transactions on Reliability, 66(3), 854 874. doi:10.1109/tr.2017.2705662

[14] N. Rani, and J. Chaudhary. **A Clustering Improved Cost Effective Approach for Mutation Testing,** Procedia Computer Science. 58. 593-602. 10.1016/j.procs.2015.08.078. (2015).

[15] A. Abuljadayel, & F. Wedyan. **An Approach for the Generation of Higher Order Mutants Using Genetic Algorithms,** I.J. Intelligent Systems and Applications, 1, 34-45. (2018) https://doi.org/10.5815/ijisa.2018.01.05

[16] A. T. Acree. **On Mutation**, PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[17] T. A. Budd. **Mutation Analysis of Program Test Data,** PhD Thesis, Yale University, New Haven, Connecticut, 1980.

[18] A. Derezinska, and M. Rudnik. **Evaluation of Mutant Sampling Criteria in Object-Oriented Mutation Testing,** Proceedings of the 2017 Federated Conference on Computer Science and Information Systems.

[19] I. Bluemke and K. Kulesza. **Reductions of Operators in Java Mutation Testing,** Advances in Intelligent Systems and Computing. 286. 2014.

[20] A. P. Mathur. **Performance, effectiveness, and reliability issues in software testing,** 1991 Proceedings the Fifteenth Annual International Computer Software & Applications Conference. doi:10.1109/cmpsac.1991.170248

[21] A. J. Offutt, G. Rothermel, & C. Zapf. An experimental evaluation of selective mutation, Proceedings of 1993 15th International Conference on Software Engineering. doi:10.1109/icse.1993.346062

[22] W. E. Wong. **On mutation and data flow**. Phd thesis Purdue University, West Lafayette (1993).

[23] W. E. Wong and A. P. Mathur. **Reducing the cost of mutation testing: An empirical study,** Journal of Systems and Software, 31(3), 185–196. 1995. https://doi.org/10.1016/0164-1212(94)00098-0

[24] S. Hussain. **Mutation Clustering**, Master's Thesis, King's College London, UK. (2008).

[25] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. C. Silva, H. L. J. Filho, & H. V. Ehrenfried. **Evaluating Different Strategies for Reduction of Mutation Testing Costs,** Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing - SAST. doi:10.1145/2993288.2993292. (2016).

[26] A. J. Offutt. **Investigations of the software testing coupling effect,** ACM Transactions on Software Engineering and Methodology, 1992.

[27] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, & A. Ventresque. **Assessing and Improving the Mutation Testing Practice of PIT,** 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST).

[28] M. Andersson. **An Experimental Evaluation of PIT's Mutation Operators.** (2017).

[29] H. Aman, R. Ibrahim. (2014). **Formalization of Transformation Rules from XML Schema to UML Class Diagram**, International Journal of Software Engineering and Its Application, *8*(12), pp.75-90, 2014.

[30] B. Mondal, B. Das, & P. Banerjee. (2014). **Formal Specification of UML Use Case Diagram – A CASL based Approach**, International Journal of Computer Science and Information Technologies, Vol. 5 (3), 2014, pp. 2713-2717.

[31] N. Ibrahim, R. Ibrahim, M.Z.Saringat, D. Mansor, & T. Herawan. (2011). **Consistency rules between UML use case and activity diagrams using logical approach.** International Journal of Software Engineering and its Applications, 5 (3), pp. 119-134, 2011.