

# Analysis of Mutation Testing using PIT and MuClipse

Anamika Frederick<sup>1</sup> Prateek Singh<sup>2</sup> Vineesh Cutting<sup>3</sup>

<sup>1</sup>M. Tech Scholar <sup>2,3</sup>Assistant Professor

<sup>1,2,3</sup>Department of Computer Science & Information Technology

<sup>1,2,3</sup>SHUATS, Prayagraj, Allahabad, India

**Abstract**— Mutation testing is an extraordinary testing technique to introduce faults in original program [11] to enhance the robustness of the code and to create better test cases. Special mutant operators are used to create large number of mutants. The result produced must be different from the original program [6]. Mutation testing requires large number of mutants [1] to be tested against the test cases being generated [6]. Thus, automatic generation of mutants are preferred to reduce the overall cost. This paper represents an efficient and simplified analysis for generating mutants using PIT tool over MuClipse tool using Eclipse IDE environment for reducing cost which is a complex problem in any software testing.

**Keywords:** Mutation Testing, Mutants, Killed, Live, PIT Tool, Muclipse Tools

## I. INTRODUCTION

Software testing is an expensive part of software development model. Drawback of software testing is that software system involves creation of large number of tests input and it is impossible to enumerate all the test inputs explicitly. To overcome this drawback and to enhance software system quality, testing criteria is used for correctness of a software system.

Mutation Testing is one of the methods of software testing to test software by applying mutation operators to original program. The first concept was given by Richard Lipton in 1971 and since then huge growth has been occurred.

A mutation operator is selected and is applied to original program, the result is a mutant program, while phenomenon is called mutation [2]. Mutant is simple program but with a small change made in the original program just by adding mutation operator to it. Once a program is modified, a set of test cases are executed against the mutant code [3]. The mutated code passes or fails the test generally depends on the testing quality.

Original program:	<pre>int large() {     if(s&gt;p)     return(s);     else     return(p); }</pre>
Mutant program:	<pre>int large() {     if(s&lt;p)     return(s);     else     return(p); }</pre>

Table I: GENERATION OF MUTANTS

In Table I, if values for s and p in one test set as s=1, p=2; then above functions in original program returns value

2 which is the actual output. Now five mutants are created one by one by replacing ">" operator by these (<, <=, >=, =, < >) operators in the if statement. For instance, when you replace ">" operator by "<", the mutant program for it is also shown in above Table I [3].

Suppose value for s and p in one test data set as s=1, p=2 then mutant returns value 1 which is different from the actual output. Hence mutant is said to be killed or dead. After executing each mutant, the result is shown in Table II [4].

Mutant	Result	Output
If s<p then	1	Killed
If s<=p then	1	Killed
If s>=p then	2	Live
If s=p then	2	Live
If s<>p then	1	Killed

Table II: KINDS OF MUTANT & THEIR RESULT

After the execution of original and mutant program, if the result of both the original and mutant program is different, the mutation is said to be killed else it lives.

The ability to find out the change in the original as well mutant program will show that the test case is sufficient enough to perform the testing. This paper represents that PIT testing technique generates lesser number of mutants and mutation score is higher as compared to MuClipse testing technique thus reducing cost factor in software testing.

The rest of the paper is organized as follows:

Section II the proposed technique for generation of mutants over PIT & MuClipse is discussed. In Section III, the result represents performance of the experiments for proposed method. Section IV represents conclusion followed by Section V that represents future scope.

## II. PROPOSED WORK

Mutation testing has always been time consuming and complex task to accomplish since there are many mutant programs that need to be generated and large number of mutation score needs to be analysed. To solve this problem, it is proposed to use automation tool as for generating and killing large numbers of mutants there by comparing different mutation tools.

New proposed method takes up the original program and generates mutants for PIT and MuClipse testing tool using Eclipse IDE (integrated development environment).

- 1) Step 1: Inputting data: First and foremost, mutants are created then line coverage and mutation coverage are analysed on eclipse using PIT. Again, mutants are created then line coverage and mutation coverage are analysed on eclipse using MuClipse this time.
- 2) Step 2: Generate separate test Cases for input original and mutated program using JUnit
- 3) Step 3: Apply the test sets on original and mutated program created by mutation testing tools separately.
- 4) Step 4: Outputting data

Each tool results with mutation score and the total number of generated mutants.

- 5) Step 5: Analysis of mutation testing tool  
Mutation tools are compared and mutation score is now recorded with no. of mutants for analysing which tool generates higher mutation score with lesser no. of mutants.

#### A. Algorithms for First Iteration:

Generating mutants with PIT tool

- 1) Configure PIT plugin on Eclipse. The system/program under test is modified by rewriting the source to introduce a single fault.
- 2) Import the program to be tested on Gradle Project Environment.
- 3) Prepare test cases for the original code/program.
- 4) Apply test cases to original code/program.
- 5) The mutants will be generated automatically using PITest library.
- 6) Code coverage and Mutation Coverage is performed [2]. The result is generated in percentage
- 7) Mutation Analysis of Result. The mutant is “killed” if test cases fail in the tests [2]. The mutant is “survived” if test cases cannot affect the behaviour of the tests.

#### B. Algorithms for second iteration:

Generating mutants with MuClipse tool

- 1) Configure MuClipse plugin on Eclipse [5]. The system/program under test is modified by rewriting the source to introduce a single fault.
- 2) Import the program/source code and unit tests in separate folders.
- 3) Select the mutation operators, generally the class-level and method-level operators [6].
- 4) Configure:
  - a) Mutants Output folder as "result".
  - b) Classes Folder as “bin”.
  - c) Test Folder as “root”.
  - d) Source Folder as “root”.
- 5) Select the class (or classes) which are targeted for the test case.
- 6) Each root node in the view of the tree that will appear is a class, underneath which the Class-Level are and Method-Level mutant nodes. The Method-Level mutants are separated by method name.

The node names correspond to the name of the mutation operator and this instance of its mutation. The test results will show the name either "killed" or "alive" depending on its status.

### III. RESULTS

The result analysed so far shows that mutation testing is introduction of fault to the original program voluntarily or involuntarily. To reduce the cost of mutation analysis various automated tools were developed. They rely heavily on the programming language used and on the set of mutation operators worked upon. The emphasis is on the mutation operators throughout the analysis [7].

The performance of running each tool on the sample programs selected in this study is shown in Table III and Table IV.

The proposed steps for implementation of automated tools makes the process far faster and easy. Traditional and class level mutants are created by ‘MuClipse’. The program P1 ‘Calc’ has the highest number of traditional mutants 37 over 51 lines of code. While the program ‘MyClass’ with LOC (Line of Code) 13 has 12 traditional mutants only. Mutants Generation depends upon various operators like arithmetic, logical or shift operators used in the program. The choice of selecting the Mutation operators on the programs depends on tester’s choice. In this paper all the traditional mutation operators are applied for the mutant creation.

Java classes	LOC	Method	Description	JUnit test coverage
P1	51	6	Contains value and functions for integer type	100%
P2	42	5	Contains value and functions for String type	100%
P3	13	1	Contains value and functions for Boolean type	100%

Table III: Junit Test Coverage Description for Specified Program

In Table IV, Mutants generated by ‘PIT’ are lesser than ‘MuClipse’. PIT applied selective traditional operators like ABS (Absolute Value Insertion), AOR (Arithmetic Operator Replacement) [12], etc. with some operators like EMM (Modifier method change), EOC (Reference comparison and content comparison replacement) [13], etc while MuClipse included all the traditional operators as well as class level operators and therefore, yielded more mutants than PIT. Mutation score of PIT was found to be a little better than that of MuClipse. PIT also runs faster.

The experiment showed PIT obtained highest mutation score. As per analysis, average mutation score using MuClipse is less than PIT. Hence, MuClipse was found to be slower because it recompiles the mutant again and again during analysis [9].

Sample Program:		P1	P2	P3	TOTAL
MuClipse	Mutants Generated	37	31	12	80
	Mutation Score (%)	89	70	75	78
	Execution Time (in sec)	3.6	6.4	5.0	15.0
PIT	Mutants Generated	22	10	6	38
	Mutation Score (%)	86	80	100	88.67
	Execution Time (in sec)	5.0	2.0	2.0	9.0

Table IV. Mutants Generation with Mutation Score

The total number of mutants generated and the average mutation score as well as the overall time of execution for each mutation tool on each test data set are being represented respectively in the Fig. 1 & Fig. 2.

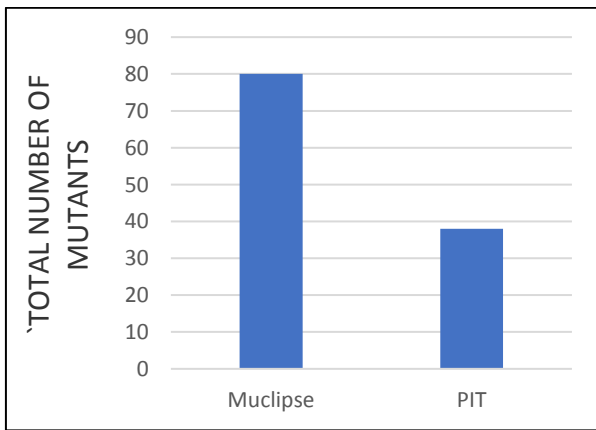


Fig. 1: Total number of mutant generations by each Automated testing tool.

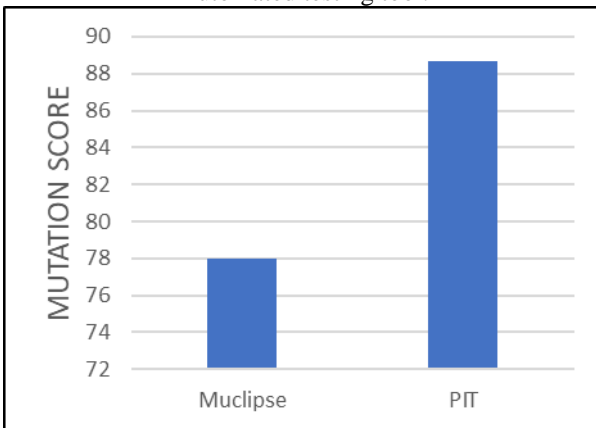


Fig. 2: Average percentage of mutants killed by each automated testing tool.

Fig. 1 & 2 demonstrate the cost in terms of mutants and mutation score for the tools [8]. It is clear that PIT killed maximum number of mutants compared MuClipse. PIT also produces a small set of mutants.

The proposed testing is simulated using java to evaluate effectiveness of a test case i.e. it should kill all the mutants though some mutant can survive.

#### IV. CONCLUSION

Mutation testing is a very powerful approach for detecting the pieces of code that are executed by running tests, though not completely tested. Mutation testing is a powerful technique to explore the behaviour of the application under test and the existing tests in connection with small code changes.

The results generated shows that large number of mutants can be generated using automation tools over manual mutant creation process. Larger the number of mutants ensures that more faults can be tested against the test class for specified programs before deploying the software or program in the real working environment [10] hence giving the opportunity to developers and testers to create better test cases in order to kill the alive mutants to perform thorough testing on program making it more robust. It is observed that PIT generated less mutants than MuClipse and also it killed more mutants while MuClipse has larger number to generated mutants to be tested. Both tools have some special features and operate on a different set of mutation operators. Thus, depending on the requirements, duration and cost of

application, an appropriate tool can be chosen using our results. It can be concluded that all the testing tools are language dependent and each tool works on a different set of mutation operators.

#### V. FUTURE SCOPE

In the near future, this work can be extended for achieving more efficiency and higher mutant generation with high mutation score. Researchers can work on implementing other automated mutation tools to compute better analysis to achieve mutation testing at lower cost.

#### REFERENCES

- [1] A.T. Acree, "On Mutation", Ph.D. thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.
- [2] Description of Mutation Testing from Wikipedia at: [https://en.wikipedia.org/wiki/Mutation\\_testing](https://en.wikipedia.org/wiki/Mutation_testing)
- [3] Yue Jia Student Member, IEEE, & Mark Harman Member, IEEE, "An Analysis and Survey of the Development of Mutation Testing", IEEE, 0098-5589/10, 2010.
- [4] David Schuler, "Assessing Test Quality", Dissertation zur Erlangung des Grades, der Naturwissenschaftlich-Technischen Fakultaten, der Universitat des Saarlandes, Saarbrucken, 2011.
- [5] Eclipse. [Online]. <https://www.eclipse.org/>
- [6] Marinos Kintis\*, Mike Papadakis\*, Andreas Papadopoulos†, Evangelos Valvis†, Nicos Malevris†, and Yves Le Traon\*, "How Effective Mutation Testing Tools Are? An Empirical Analysis of Java Mutation Testing Tools with Manual Analysis and Real Faults", Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg, ({marinos.kintis, michail.papadakis, yves.letraon}@uni.lu) †Department of Informatics, Athens University of Economics and Business, Greece, ({p3100148, p3130019, ngm}@aueb.gr)
- [7] Hu, J., Li, N., & Offutt, J. "An analysis of oo mutation operators". In Mutation '11: Proceedings of the 6th International Workshop on Mutation Analysis, 2011.
- [8] A. M. R. Vincenzi, A. S. Simão, M. E. Delamaro & J. C. Maldonado, "Muta-Pro: Towards the definition of a mutation testing process", Journal of the Brazilian Computer Society, volume 12, pages49–61, 2006.
- [9] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, & F.G. Sayward, "Mutation Analysis Technique Report", GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, Georgia, 1979.
- [10] R.T. Alexander, J.M. Bieman, S. Ghosh, and B. Ji, "Mutation of Java Objects in Proceedings of the 13<sup>th</sup> International Symposium on software Reliability Engineering (ISSRE'02)", 351 Annapolis, Maryland, IEEE computer Society, pages 34, 12-15 November 2002.
- [11] P. Ammann and J. Offutt, "Introduction to Software Testing", Cambridge University Press, 2008.
- [12] J.S. Bradbury, J.R. Cord, & J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0), Proc. Of the 2<sup>nd</sup> workshop on Mutation Analysis", Pages 83-92, 2006.

- [13] Bieman, J., Ghosh, S., and Alexander, R.T., (2001) “A Technique for Mutation of Java Object In Proceedings of the 16<sup>th</sup> IEEE International Conference on Automated Software Engineering (ASE’01)”, San Diego, California, pages 337, 26-29 November 2001.

