Journal of Analysis and Computation (JAC)
(An International Peer Reviewed Journal), www.ijaconline.com, ISSN 0973-2861
Volume XII, Issue VII, July-Dec 2018

ANALYSIS OF STRONG AND SELECTIVE MUTATION TESTING TECHNIQUES FOR PYTHON PROGRAM

Sandeep U Kadam[1] , Sunil G. Dambhare[2], Chaitali R Shewale[3], Rupesh J Patil[4]
Associate Professor, Bhivarabai Sawant College of Engineering and Research, Pune, India[1]
Professor, Dr. D. Y. Patil Institute of Engineering Management and Research, Pune, India[2]
Assistant Professor, Keystone College of Engineering, Pune, India[3]
Principal Navsahyadri Grooup of Institutes Faculty of Engineering, Pune , India[4]

ABSTRACT
Software Testing is the process of ensuring quality of software through detailed investigating with objective of finding the faults in it. Software testing is associated with the Test Suit which is constituted of Test Cases for checking correctness of the software system. It is very important to have stronger Test Suit and adequacy in terms of Test Cases for ensuring quality of Software. Mutation Testing is white box testing method in which test suit is assessed for its quality. Many mutation testing techniques are proposed by researchers over last few decades for testing the test cases. The Proposed study presents the detailed analysis of selective and strong mutation testing techniques applied to the python programs from differ applications. Mutant generation is one of the most important task in mutation testing. Python is one of the most popular programing languages. The experimental analysis is carried out using open source MutPy for creating mutants with different mutation operators.

Keywords
Mutation Testing, Test Case, Mutant, Mutation operators, Mutation Score

1. Introduction

Software Testing is the process to ensure software product without flaws by testing the software product to meet expected requirements. The main purpose of software testing is to identify the errors or bugs and the requirements that are not in line with expected outcomes. White box or black box testing techniques can be chosen appropriately for testing the software. Software testing can identify the defects in the software in initial stages and can be resolved prior to delivery of the. The precise and effective software testing ensures high reliability, security and performance that continuously saves time, cost-effectiveness and customer satisfaction.
In software testing, test cases or test suites plays measure role in ensuring quality of software and the final performance of test cases is measured based on their ability to detect actual errors, in other words how many errors the test can detect while running. Unfortunately, this measure does not always work because one needs to know about the effectiveness of test cases through

appropriate measurement prior to its execution [1]. Mutation testing is the way to measure the abilities of the test suits in detecting errors. Mutation testing is a white box testing method in software testing where we insert errors purposely into a program (under test) to verify whether the existing test case can detect the error or not. In this testing, the mutant of the program is created by making some modifications to the original program. The test case is executed for original program and mutant program. If no change in output observed, then the test case is weak and may fail to detect the errors in software product. Mutants are errors that the programmer is likely to have created by accident. Many times the majority of software flaws created by skilled programmers are caused by minor grammatical errors. As a result, each mutant introduces a minor modification, i.e. a flaw, in the system as compared to the original. Sometime in mutation testing complicated faults are coupled with simpler faults in such a way that test cases that identify all the basic faults will also find the complicated faults. Mutated versions are made by using mutation operators, which are rules that are applied to the system in order to create mutants. These are simple syntactic or semantic transformation rules like deleting an assignment expression, replacing or inserting new operators to construct syntactically legal expressions, etc.

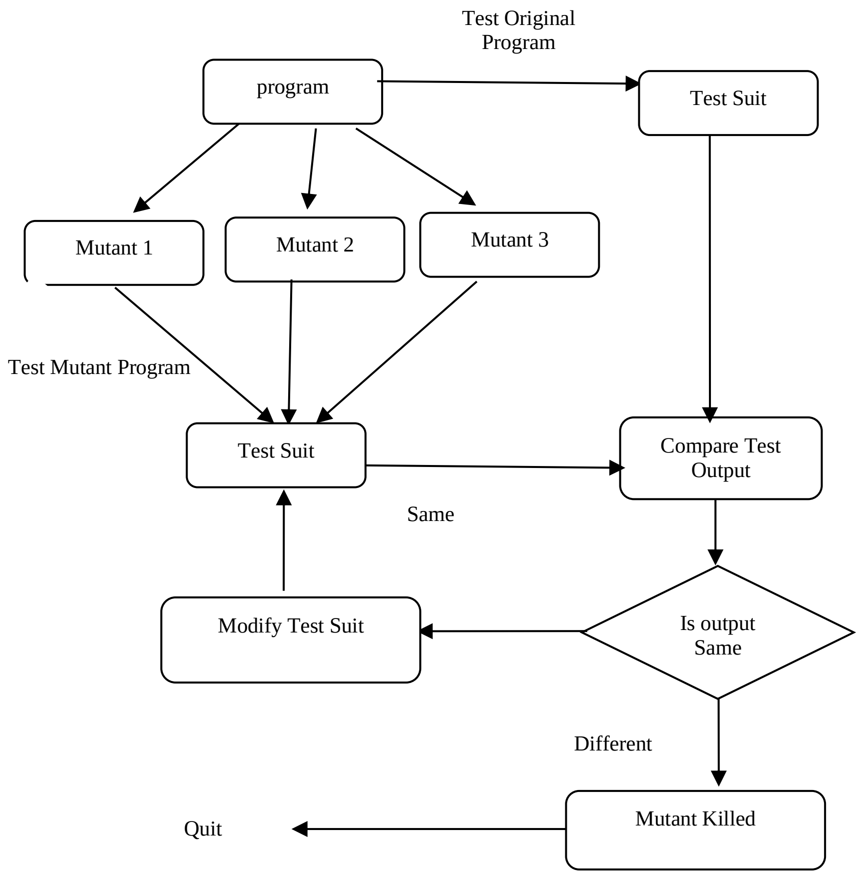

Figure 1: Process of Mutation Testing

Mutation Testing can accurately determine the accuracy of a test sample, but it still has a number of flaws. The high computational cost of performing the enormous number of mutants against a test set is one issue that prevents Mutation Testing from being a realistic testing technique. The other issues revolve around the amount of human effort required to use Mutation Testing. Mutation testing, on the other hand, is non-effective because it is time-consuming, which can lead to a rise in the number of test cases. Furthermore, due to the ambiguity of mutant equivalence, [3, 4] detecting identical mutants usually necessitates additional human effort.

Mutation Testing analysis process is described in Figure 1 where the original program is changed to different version by making small changes to it with the help of mutation operators supported by the specific programing language. The original program and mutated programs are executed for same Test Suit. If the original program and Mutant Program gives different output results, then mutant is said to be killed and test Suit is strong and if the output result is same then Test suit is said to be weak and need to be reviewed again as it may fail to detect the bug in the software product.

Python is one of the eight most popular programming languages in the recent decade. It's possible that as a dynamically typed language, it'll be more sensitive to little programming errors and more difficult to test. Python is object oriented programming language and supports the features of object oriented programming language. The ability to test Python programmes effectively is essential. This study attempted to use the object oriented mutation testing approach to python programs for analyzing better approach of Mutation Testing which can be applied to python programs.

2. Literature Review

In terms of mutation operators, mutation testing is a language-dependent methodology, which brings special challenges in the case of C++. Pedro Delgado-P´erez, et. al. proposed the work to define a set of C++ mutation operators and to build a practical and comprehensive solution for automating mutation in this context. This technique is based on traversing the abstract syntax tree [5]. To demonstrate the technique's usefulness, these operators were examined through a well-constructed experiment using several OO applications.

Evolutionary mutation testing approach is proposed in which the term "evolutionary mutation testing" refers to a combination of the terms "evolutionary testing" and "mutation testing." It is an attempt to combine the strengths of evolutionary testing with mutation testing to automate the test case generation process and reduce mutation testing's computing overhead. The state of an object is important in testing, and control flow information is also useful for learning about a program's behavior. As part of test case fitness, it is suggested to have fitness function that incorporates object state and control flow information. The evolutionary mutation testing procedure is improved by using object state fitness, control flow information, two-way crossover, and adaptable mutation [6]. The approach proposed for the same number of mutants, rank-based mutant selection produced even better results: 98.87 percent mutation score and 5.72 percent and percentage of test cases loss [7]. As a result, in the case of class mutation operators, selecting mutants from all mutation operators

in a rank-based approach has proven to be more beneficial than operator-based selective mutation. For mutant selection, the proposed rank-based strategy outperformed the random methodology.
Recent research in area of Mutation has focused on developing scalable and practical technologies that can help to bring mutation testing into the mainstream of industry and everyday life. The discovery of equivalent and redundant mutants is one of the most difficult open challenges in mutation. Unfortunately, there is no clear theory or agreement on which mutant kinds and instances should be used. The fact that most available tools are confined to a restricted number of mutant operators is constraining and arbitrary to some extent. Most past research on first order mutation has been limited due to a lack of a clear idea on which mutants are of some use [8].

The goal-oriented mutation testing with focal method proposes that it is best to test a specific method in a test case that is specifically designed to test that method, rather than a test case that just happens to call the method in one of its routines, for improved ability of the test suite for diagnosis and maintenance. The focus on those test that are designed to test and assure the quality of certain techniques using focused methods. This will improve not just the overall effectiveness of the test suite to find errors, but also the quality of each method and its related test cases [11].
The majority of mutants that the methods fail to kill, revealing flaws in the methods' test sets, are linked to programme coverage. Because none of the OO methods need statement coverage, it looks that they do not execute all of the program's statements. Traditional coverage methods, like as control-flow testing, may easily address this flaw. Some coverage methods should be utilized with the OO approaches in order for them to be successful. The Class Mutation results also reveal that, although being stated to be appropriate for OO systems, the OO methods are ineffective when dealing with a few OO characteristics. Assessing the efficacy of OO testing methodologies appears to be a critical yet underutilized subject. The mutation operators' fault models could serve as a starting point for a theoretical evaluation of OO methods' fault detection capacity [12].

3. Mutation Testing and Mutation Operators in Python

A static analysis can verify a code substitution defined by a standard mutation operator in strongly typed programming languages. At compilation time, the types of data, variables, and operators, as well as type consistency rules, are known. As a result, we may be confident that an altered programme will be appropriately compiled, and that a mutation will not result in type-related errors at run-time. Some object-oriented mutation operators are more difficult to employ since they are dependent on a variety of factors, such as other classes in an inheritance chain. It is feasible to avoid erroneous code updates by checking appropriate accuracy criteria.
A well-developed test suite is mostly required by a developing team in order to offer high-quality software projects. Code coverage and test suite evaluation are two methodologies that seek to evaluate test suites in some way. Testing for mutations. The percentage of source code that is covered is referred to as code coverage. When a test suite is run, a programme executes Testing for mutations The effectiveness of the test suite is measured. More development teams use code

coverage to a greater extent than Testing for mutations. Code coverage is tracked throughout a project.

Mutation testing seeks to create a set of test cases that are sufficient in finding almost all defects in a programme. It is used to put a test case to the test and kill all of the mutants in it. With a high score, a test case is deemed to be acceptable if it can kill all nonequivalent mutations. Mutation score is defined as the total number of killed mutants over non-equivalent mutants. The test suite must successfully kill all mutants in order to achieve the highest mutation score.

$$\boxed{\text{Mutation Score} = \frac{\text{Number of Mutants Killed}}{\text{Number of Non- Equivalent}}}$$

Figure 3: Mutation Score

As a result, in order to achieve the best possible score, all non-equivalent mutants must be identified. Such mutants merely serve to increase the computational cost of mutation testing. They don't determine whether or not a given test case is effective in detecting programme flaws. In addition, determining whether a mutant is equivalent to the original mutant is theoretically difficult.

In many circumstances, Java and C# are keywords that can be utilized or not. In Python, the equivalent keyword self is required. As a result, the operator JTI deletes this type of data. Following is an example of Mutant Program

```
def add (a, b):
return a + b;
```

```
def add (a, b):
return a * b;
```

Figure 2: Mutant Program

This proposed study uses MutPy tool which is a mutation testing tool for source code of Python version 3.3 and up. MutPy features a standard unit test module, generates YAML/HTML reports, and outputs in a colorful manner. It uses AST (Abstract Syntax Tree) level mutation, high order mutations (HOM) and code coverage analysis.

In analyzing, comparing, and enhancing the quality of a test suite, mutation testing approaches are critical. Nonetheless, the value of mutation testing is determined by the mutants utilized in the analysis. Through the use of preset mutation operators, these mutants are created from the original programme. Mutation operator is a rule that substitutes sections of the source code in order to conduct syntactic changes on the programme. Researchers have built and created a collection of mutation operators to support diverse programming languages such as Python, because mutation is always based on mutation operators. Mutation testing relies heavily on the quality of mutation operators.

The main purpose of using the mutation operator is to build a large number of incorrect versions of the original programme 10]. Mutants are the name given to these variants. The mutation operator is used to the original source code to create a mutant. An operator is a rule that substitutes sections of source code in order to change the program's syntactic structure [9].

The production of mutants is depicted in Figure 2. The effectiveness of mutation testing is determined by the effective mutants' selection. The test engineer runs the mutants through a series of test cases and compares their behavior to that of the original software in order to spot any errors. The mutation score [9] is the percentage of mutants destroyed by the test cases.

Many researchers have proposed various mutation operators for different programming languages which are also supported by mutation testing tools developed for that programming language. This study focuses on python, with a particular emphasis on the operators that deal with object-oriented features. The mutation operators can be categorized in to two categories as structural mutation operators and object oriented mutation operators. The mutation operators mentioned below are considered in this study. These are supported by MutPy Tool.

A. Object Oriented Mutation Operators

EHD- Exception Handler Deletion
EXS- Exception Swallowing
IHD- Hiding Variable Deletion
IOD- Overriding Method Deletion
IOP- Overridden Method Calling Position Change
SCD- Super Calling Deletion
SCI- Super Calling Insertion
CDI- Class Method Decorator Insertion
SDI- Static Method Decorator Insertion
SVD- Self Variable Deletion
DDL- Decorator Deletion

B. Structural Mutation Operators

AOD- Arithmetic Operator Deletion
AOR- Arithmetic Operator Replacement
ASR- Assignment Operator Replacement
BCR- Break Continue Replacement
COD- Conditional Operator Deletion
COI- Conditional Operator Insertion
CRP- Constant Replacement
LCR- Logical Connector Replacement
LOD- Logical Operator Deletion
LOR- Logical Operator Replacement
ROR- Relational Operator Replacement
SIR- Slice Index Remove
OIL- One Iteration Loop

RIL- Reverse Iteration Loop
SDL- Statement Deletion
ZIL- Zero Iteration Loop
This study uses five different python projects available for analysis of mutation testing by using strong and selective mutation testing techniques. The experimental results of mutation testing for these five projects with the application of four different mutation techniques object oriented level operators, structural level operators [13], all operators and random sampling [11] are compared.

4 Experimental Analysis

Development of mutation is critical and important stage in mutation testing and it consumes resources at large. Many automation mutant creation tools are proposed and developed to make the process more efficient. MutPy [18], a mutation-testing tool is used in this study for creating mutants with the help of various mutation operators supported by it. MutPy was developed as academic project at Warsaw University of Technology's Institute of Computer Science, Faculty of Electronics and Information Technology. MutPy version 0.2, a mutation testing tool based on Python, was released in 2011. It was based on the manipulation of an abstract syntax tree (AST). Latter there is development up to MutPy version 0.6.1 with updated mutant production algorithm and mutant detection technique. The set of mutation operators supported by MutPy is discussed in section 3. MutPy reduces the time required for mutation creation by more than 50%. The proposed study analyses the mutation testing score with four different techniques mutation testing with structural level operators, mutation testing with object oriented level operators, mutation testing with all operators and Mutation testing with random sampling operators. Five different python applications developed by students as part of academic project are used for this proposed analysis. The mutation testing score is calculated for each of the module of the python application and mean mutation score is calculated for each application. This process of Mutation Testing is repeated for all the python applications. The results obtained are described in the
table provided in this section.

Table 1: Mutation Testing with Structural Level Operators

Project Name	Project Details		Mutation Testing Results			
	Name of Module	Number of methods for Test	Killed mutants	Number of Live Mutants	Mutation Score	Mean Score
Music Player	Song Inventory	9	94	88	51	67.33
	Player	8	69	24	74	
	ThemeSelector	5	84	25	77	
Railway Booking System	StationMaster	6	36	5	87	89
	SeatMaster	4	22	6	78	
	PassangerMaster	3	27	0	100	
	ScheduleMaster	9	48	7	87	
	BookingMaster	4	30	2	93	
Quiz Application	Question	9	47	132	28	51
	Student	11	68	36	69	
	Teacher	4	9	2	81	

	Subject	3	17	47	26	
Chatbot	Dictionary	9	4	10	30	24.66
	Query	6	3	127	2	
	User	3	3	4	42	
Library Management System	BookInventory	5	76	45	62	65.33
	BookBank	4	56	28	66	
	LibraryUser	3	28	13	68	

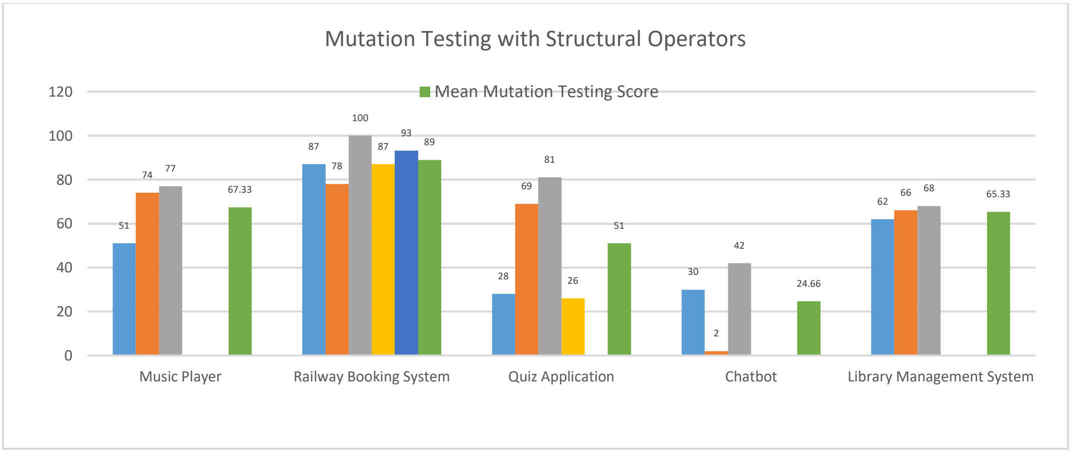

Figure 4: Mutation Testing with Structural Operators

Structural level operators are applied for creating mutants for the programs of five different applications selected for experimentation. It is observed from the experimentation result analysis Railway Booking system application is having highest mean mutation score of about 89%. The bar graph shows the comparative analysis of mutation score for every application and mutation score is calculated for each module of the application. Mean Mutation score is also calculated for each application and shown in bar graph.

Table 2: Mutation Testing with Object oriented level Operators

Project Name	Project Details		Mutation Testing Results			
	Name of Module	Number of methods for Test	Killed mutants	Number of Live Mutants	Mutation Score	Mean Score
Music Player	Song Inventory	9	34	48	42	36.66
	Player	8	16	27	37	
	ThemeSelector	5	13	28	31	
Railway Booking System	StationMaster	6	41	7	85	85.8
	SeatMaster	4	18	4	81	
	PassangerMaster	3	23	2	92	
	ScheduleMaster	9	49	9	84	

	BookingMaster	4	14	4	87	
Quiz Application	Question	9	27	92	22	48
	Student	11	25	27	48	
	Teacher	4	11	2	84	
	Subject	3	24	38	38	
Chatbot	Dictionary	9	11	51	17	13.33
	Query	6	1	7	12	
	User	3	4	32	11	
Library Management System	BookInventory	5	1	9	10	10.33
	BookBank	4	2	18	10	
	LibraryUser	3	7	62	11	

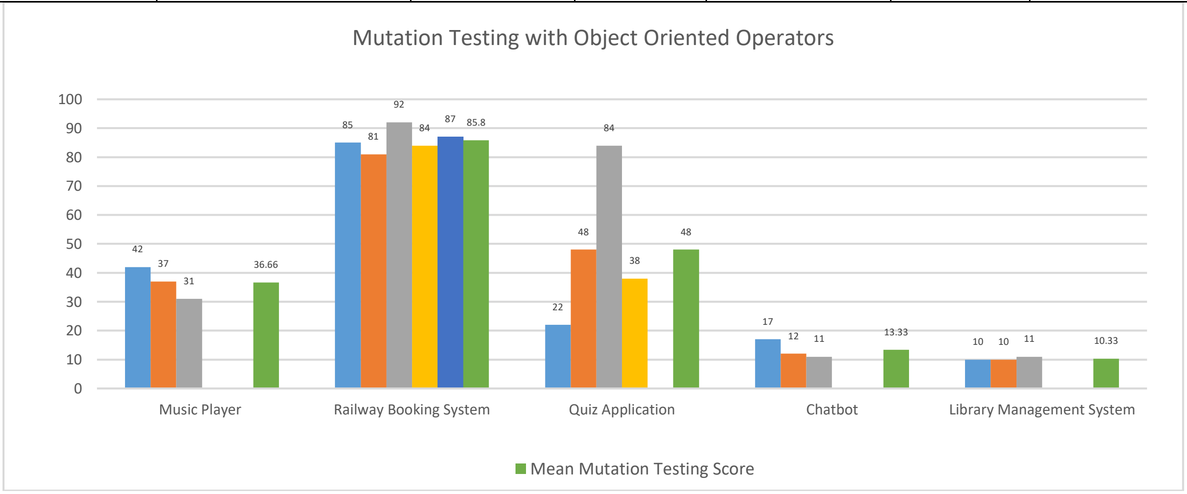

Figure 5: Mutation Testing with Object oriented level Operators

Table 3: Mutation Testing with All Operators

Project Name	Project Details		Mutation Testing Results			
	Name of Module	Number of methods for Test	Killed mutants	Number of Live Mutants	Mutation Score	Mean Score
Music Player	Song Inventory	9	94	48	51	**73**
	Player	8	45	14	76	
	ThemeSelector	5	76	23	77	
Railway Booking System	StationMaster	6	46	4	92	**92.5**
	SeatMaster	4	22	2	91	
	PassangerMaster	3	6	0	100	
	ScheduleMaster	9	48	7	87	
	BookingMaster	4	30	2	93	
Quiz Application	Question	9	57	86	39	**58**
	Student	11	58	14	80	
	Teacher	4	11	2	84	

	Subject	3	10	25	28	
Chabot	Dictionary	9	4	10	30	**24**
	Query	6	3	127	2	
	User	3	3	4	42	
Library Management System	BookInventory	5	53	22	70	**69**
	BookBank	4	34	14	68	
	LibraryUser	3	27	9	71	

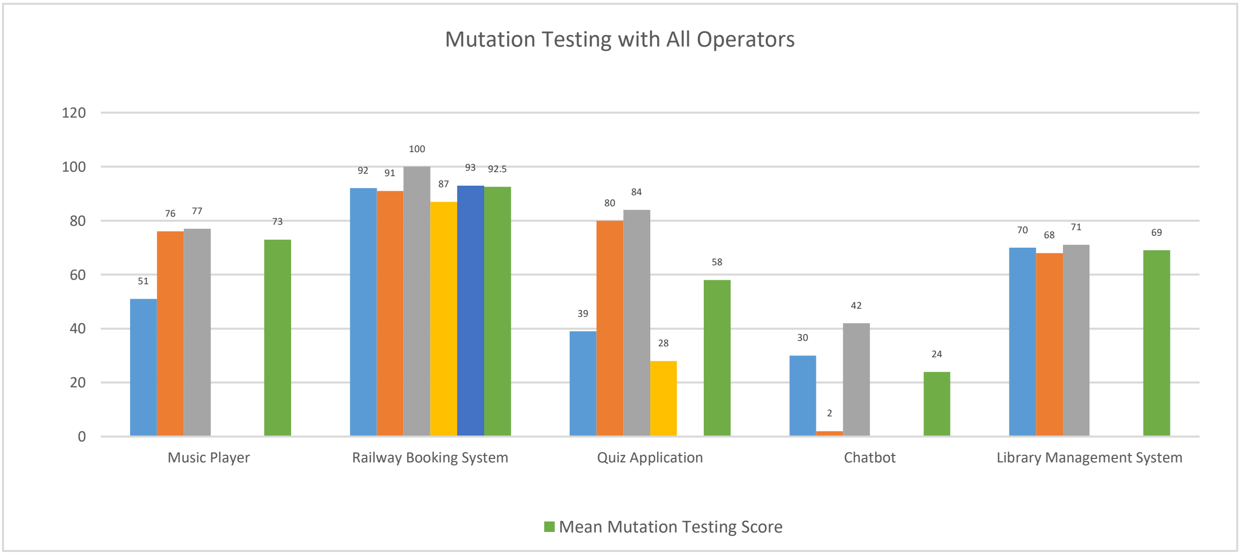

Figure 6: Mutation Testing with All Operator

Similarly, mutation score and mean mutation score is calculated for remaining methods using object oriented operators, all operators i.e. structural and object oriented operators and finally for randomly selected operators. Mutation score is recorded and bar graphs are plotted for all recordings. The railway booking system application has maximum mutation and mean mutation score in all the methods selected here for mutation testing. The comparison of mutation score is carried out in two ways one way is to compare the score of mutation testing of different modules in same application and second way is to compare mutation testing score of different applications. This gave the bigger perspective to predict better method for mutation testing.

Table 4: Mutation Testing with Random Operator

Project Name	Project Details		Mutation Testing Results			
	Name of Module	Number of methods for Test	Killed mutants	Number of Live Mutants	Mutation Score	Mean Score
Music Player	Song Inventory	9	48	21	69	70.33
	Player	8	23	10	71	
	ThemeSelector	5	69	27	71	
Railway Booking System	StationMaster	6	36	4	90	

	SeatMaster	4	23	3	92	
	PassangerMaster	3	12	1	92	91
	ScheduleMaster	9	48	5	90	
	BookingMaster	4	30	2	93	
	Question	9	25	21	54	
Quiz Application	Student	11	54	43	55	53
	Teacher	4	11	2	84	
	Subject	3	10	25	28	
	Dictionary	9	14	32	30	
Chatbot	Query	6	3	13	18	22.33
	User	3	7	29	19	
	BookInventory	5	24	11	68	
Library Management System	BookBank	4	34	20	62	
	LibraryUser	9	48	21	69	65

The experimental results analysis of mutation testing with four different mutation testing techniques structural operator, object-oriented operator, all operator and random operator on five different python applications shows that the all operator method gives better mutation score and mean mutation score for all the selected python applications.

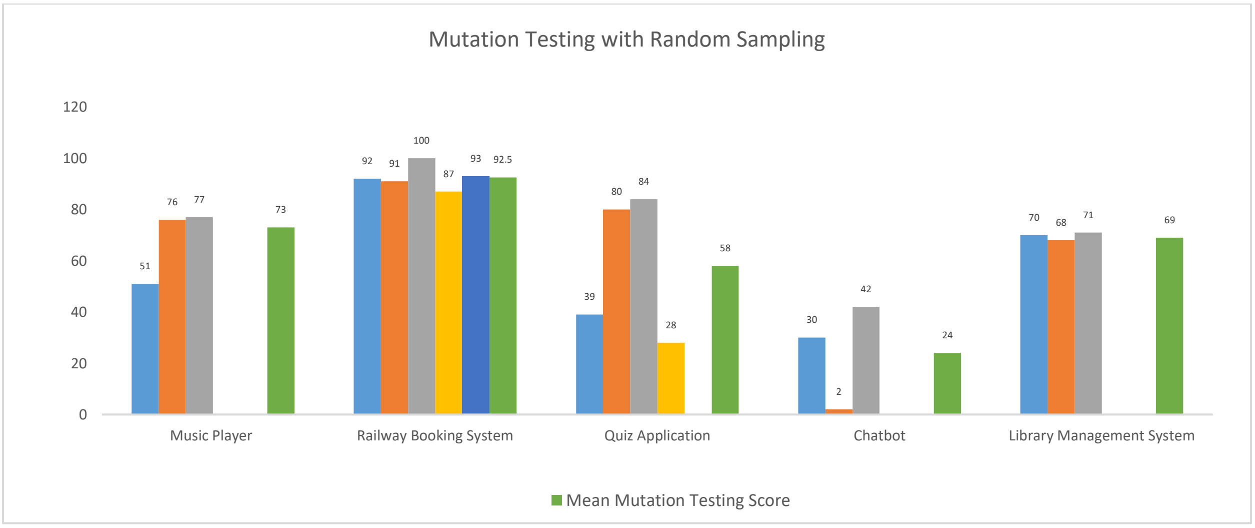

Figure 7: Mutation Testing with Random Operator

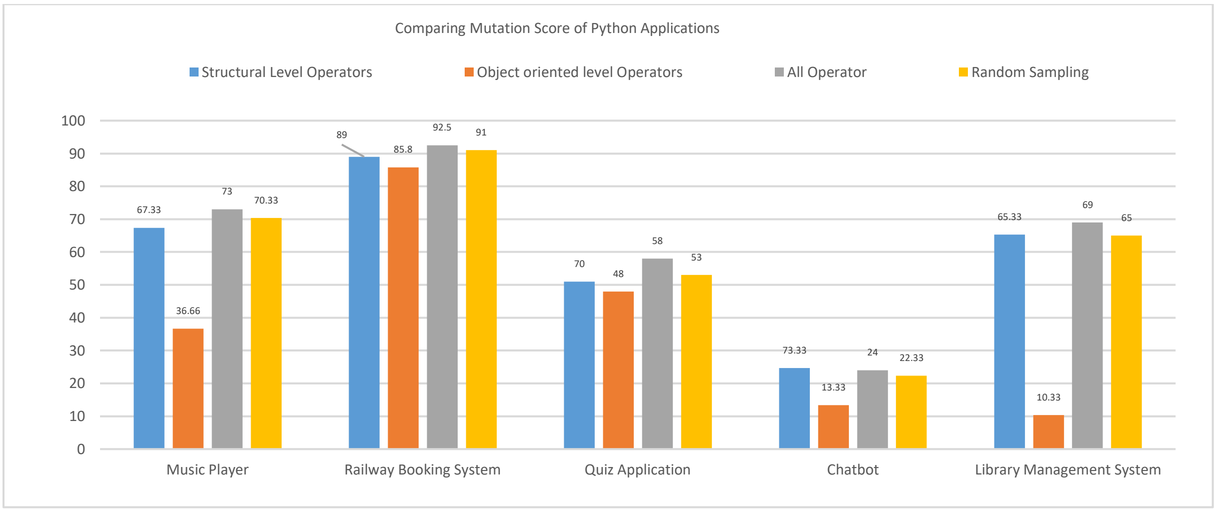

Figure 8: Compare Mutation Score of Python Applications with different methods

5. Conclusion and Future Work

This experimental analysis of mutation testing strategies for python program indicates that the all operator strategy for mutation testing of python program is most effective strategy as it kills maximum mutants and presents the adequacy of test cases for python programming language applications. Also this strategy works more effective for small and middle sized python programs. Strong mutation testing strategy like using all operator method for creating mutants becomes very complex and time consuming when the size of the program increases. It is expected that the strategy should adopt the policy so as to minimize the cost of equivalent mutants in future.

References

[1] Tanzeem Bin Noor and Hadi Hemmati. 2017. Studying Test Case Failure, Prediction for Test Case Prioritization. In Proceedings of PROMISE , Toronto, Canada, November 8, 2017, 10 pages.

[2] R. A. DeMilloand A. J.Offutt, ―Experimental results from an automatic test casegenerator‖,ACM Transactions Software Engineering Methodology,Vol. 2, Issue 2, pp. 109 –127, April 1993.

[3] T.A. Budd and D. Angluin, “Two Notions of Correctness and Their Relation to Testing,” Acta Informatica, vol. 18, no. 1, pp. 31- 45, Mar. 1982.

[4] E.J. Weyuker, “On Testing Non-Testable Programs,” The Computer J., vol. 25, pp. 456-470, 1982.

[5] Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J.J. et al. Class mutation operators for C++ object-oriented systems. Ann. Telecommun. 70, 137–148 (2015).

[6] M. B. Bashir and A. Nadeem, "Improved Genetic Algorithm to Reduce Mutation Testing Cost," in *IEEE Access*, vol. 5, pp. 3657-3674, 2017

[7] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, Juan José Domínguez-Jiménez, “Assessment of class mutation operators for C++ with the MuCPP mutation system” , Information and Software Technology, Volume 81,Pages 169-184, Elsevier,2017

[8] Henard, Christopher & Papadakis, Mike & Le Traon, Yves. (2014). Mutation-Based Generation of Software Product Line Test Configurations. 92-106. 10.1007/978-3-319-09940-8_7.

[9] Mresa, Elfurjani Sassi and Leonardo Bottaci. “Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study.” *Softw. Test. Verification Reliab.* 9 (1999): 205-232.

[10] Lingming Zhang, Darko Marinov, Sarfraz Khurshid1, " Faster Mutation Testing Inspired by Test Prioritization and Reduction", ISSTA 2013: Proceedings of the 2013 International Symposium on Software Testing and AnalysisJuly 2013 Pages 235–245

[11] Sten Vercammen, Mohammad Ghafari, Serge Demeyer, Markus Borg," Goal-Oriented Mutation Testing with Focal Methods", A-TEST 2018: Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and EvaluationNovember 2018 Pages 23–30

[12] Kim, Sun-Woo, John A. Clark and John A. McDermid. "Investigating the effectiveness of object-oriented testing strategies using the mutation method." *Software Testing* 11 (2001): n. pag.

[13] Derezinska, Anna & Hałas, Konrad. (2014). Analysis of Mutation Operators for the Python Language. Advances in Intelligent Systems and Computing. 286.

[14] Derezińska, A., Hałas, K.: Operators for Mutation Testing of Python Programs. Res. Rep. 2014, Inst. of Comp. Science Warsaw Univ. of Technology (2014)

M. E. Delamaro and J. C. Maldonado. Proteum – A tool for the assessment of test adequacy for C programs. Proceedings of the Conference on Performability in Computing Systems, pages 75–95, July 1996

R. A. DeMilloand A. J.Offutt, ―Experimental results from an automatic test casegenerator‖,ACM Transactions Software Engineering Methodology,Vol. 2, Issue 2, pp. 109 –127, April 1993.

[15] A. Derezinska and K. Halas, "Experimental Evaluation of Mutation Testing Approaches to Python Programs," *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, 2014, pp. 156-164

[16] Wu, Xiaoxue & Zheng, Wei & Shi, Zhao & Wang, Zehai & Cao, Lixin & Mu, Dejun. (2018). Concurrency Bug-Oriented Mutation Operators Design for Java. 364-369. 10.1109/PIC.2018.8706335.

[17] Devroey, Xavier & Perrouin, Gilles & Papadakis, Mike & Legay, Axel & Schobbens, Pierre Yves & Heymans, Patrick. (2016). Featured model-based mutation analysis. 655-666. 10.1145/2884781.2884821.