Semantic Mutation Testing Tool in Python

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering

in

Software Engineering

Submitted By

Ramil Gupta

(Roll No. 801331022)

Under the supervision of

Dr. Ajay Kumar

Assistant Professor

(CSED)

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

THAPAR UNIVERSITY

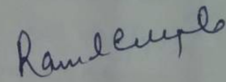
PATIALA – 147004

July 2015

CERTIFICATE

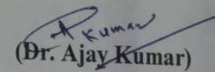
I hereby certify that the work which is being presented in the thesis entitled," *Semantic Mutation Testing Tool in Python"*, in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala; is an authentic record of my own work carried out under the supervision of *Dr. Ajay Kumar* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.



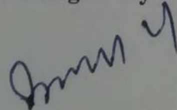
(Ramil Gupta)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

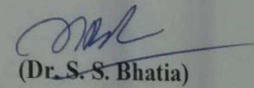


(Dr. Ajay Kumar)
Assistant Professor
Thapar University
Patiala

Countersigned by



(Dr. Deepak Garg)
Head of Department
Computer Science and Engineering Department
Thapar University
Patiala



(Dr. S. S. Bhatia)
Dean(Academic Affairs)
Thapar University
Patiala

Abstract

One of the effective techniques for testing is mutation testing. Mutant can be created by changing the syntax of a program. To distinguish the mutant from the original program, an effective test suite is required. The Mutation testing is a testing method aimed at assessing/improving the adequacy of test suites and estimating the number of faults present in systems under test.

The mutations can be applied to the source code and the semantics of the language.The mutations of the semantics of the language represent possible misunderstandings of the description language and thus capture a different class of faults. Since the likely misunderstandings are highly context dependent, this context should be used to determine which semantic mutants should be produced. The approach is illustrated through examples with state charts and code in python. In addition, a semantic mutation testing tool for Python is proposed.

ACKNOWLEDGMENT

First of all, I am extremely thankful to my respective guide ***Dr. Deepak Garg,*** Associate professor, CSED, Thapar University for his valuable guidance, advice, motivation, encouragement, moral support, sincere effort and positive attitude with which he solved my queries and provide delightful ambiance for learning, exploring and making this thesis possible. He always set high goals for me and help me to find the right direction to achieve those goals. It has been a great experience to work under his guidance.

I am also heartily thankful to ***Dr. Ajay Kumar***, Assistant Professor, CSED, Thapar University for motivation and guidance that was very helpful to achieve my goals.

I would like to thank my family members and my friends who are dearest and precious to me for their love, encouragement, blessings and support in all respects. Most importantly, none of this would have been possible without the love and patience of my family. To my brother and friends for showing me right direction. They are constant source of love, concern, support and strength for me all these years.

Finally I would like to thank management of Thapar University for proving a great platform for learning, not just for academics but also for sports and many other creative things.

Ramil Gupta

801331022

ME (SE)

Table of Contents

CERTIFICATE....................................... **Error! Bookmark not defined.**

Abstract....................................... ii

ACKNOWLEDGMENT....................................... iii

List of Figures vi

List of Tables....................................... vii

Chapter 1 Introduction....................................... 1

1.1 Software Testing 1

1.2 Purpose of software testing....................................... 2

1.3 Testing Process 3

1.3.1 Planning and Control: 4

1.3.2 Analysis and Design: 5

1.3.3 Implementation and Execution:....................................... 5

1.3.5 Activities of test Closure:....................................... 7

1.4.1 Functional Technique and Structural Technique....................................... 8

1.5 Software Testing - Methods 8

1.5.1 Black-Box Testing....................................... 8

1.5.2 White-Box Testing 9

1.5.3 Grey-Box Testing 9

1.6 Mutation Testing 10

1.7 Mutation Operators 13

1.7.1 Mutation Score (MS) 13

1.7.2 Arithmetic Operators 15

1.7.3 Relational Operators 15

1.7.4 Conditional Operators....................................... 15

1.7.5 Shift Operators 16

1.7.6Assignment Operators 16

1.8 Semantic mutation testing....................................... 16

1.9 Thesis Outline 17
Chapter 2 Literature Survey 18
Chapter 3 Problem Statement 24
3.1 Dissertation Objectives 24
3.2 Experimental Setup 24
3.2.1 PYCharm IDE 24
3.2.2 Python 25
3.2.3 Unit Test: 25
Chapter 4 Implementation 27
4.1 Working of Semantic Mutation Testing 27
4.2 Proposed Technique 28
4.3 System Architecture and Implementation 30
4.4 Results and Discussions 39
Chapter 5 Conclusion and Future scope 41
5.1 Conclusion 41
5.2 Future Work 42
Chapter 6 References 43
List of publication 49
Video Link 50

List of Figures

Figure 1.1: Testing Process 4
Figure 1.2 : Mutation testing 11
Figure 1.3 : Mutation Testing Process 14
Figure 4.4 : Working Of Semantic Mutation Testing 28
Figure 4.2 : Selecting the source program 34
Figure 4.3 : System Architecture……………………………………………………….39
Figure 4.4 : Selecting source program.………………………………………………… 39
Figure 4.5 :List of available programs ....…34
Figure 4.6 : Possible mutants to be applied to the selected source program.....................35
Figure 4.7 : Selecting the mutant...35
Figure 4.8 : Original Program 36
Figure 4.9 : Mutant Program...36
Figure 4.10 : Output in Mutant Executor- Failed Test Cases 37
Figure 4.11 : Output of Mutant Executor – Detailed View of Unit Test..........................38
Figure 4.12 : Output of Mutant Executor: Summary of Test Cases Result 38

List of Tables

Table 4.1 Set of Programs 36
Table 4.2: Traditional Mutant 32
Table 4.3 : Semantic Mutants..32
Table 4.4 : Efficiency of test suites in MST 39
Table 4.5 : Efficiency of test suites in Traditional Mutation testing. 39

Chapter 1 Introduction

This chapter has the basic introduction of software and mutation testing. It also has a brief introduction about the conditions, types, examples and the operators of mutation testing.

1.1 Software Testing

Software testing is the process of execution of a program or application with the intent of finding the bugs in the software. Software testing can also be stated as the process of validation and verification that a software program or application or product is working as expected or not:

- Meets the business and the technical requirements that guided its design and development.
- Works as per the expectation or not.
- Can be implemented with the same characteristic.

Another definition of testing is: Testing is the process of evaluating a system or its component(s) with the intent of finding whether it satisfies the specified requirement(s) or not. Testing is the execution of a system to identify any gaps, errors, or missing requirements in contrary to the actual requirements.

The definition of Software testing can be divided into the following parts:

- **Process:** Testing is a process rather than a single activity.
- **All life Cycle Activities:** Testing is a process that takes place throughout the Software Development life Cycle (SDLC) [2] [3].

Testing performed by a developer on completion of the code is also categorized as testing. There are two classes of troubles in Computer Software: faults or failures [l]. Fault is a condition that causes software to fail to perform its required function. The error is the difference between Actual and the Expected output. Failure is the in ability of

a system or a part of a system to perform required function as per its specification. Software testing plays a very important role due to the following reasons:

- Software testing is required to identify the defects and errors that were made during the coding phase of SDLC.
- It is essential since it assures the reliability of the software as per the guided specs.
- It is the most substantial part in ensuring that the product is superlative i.e. of very high quality. This helps to build the customers confidence in the service provider and in the longer run helps to retain the customer.
- Testing is compulsory for providing the facilities to the clients, for instance, the delivery of a superlative product or application that should have lesser maintenance expenditure and therefore results are of higher accuracy, consistency, and reliability.
- Effective performance of software application or product can only be achieved with the help of testing.
- It is mandatory for the application not to result in any sorts of failure since it can be it can lead to extravagant costs of the project.
- To be in business, you need to test.

1.2 Purpose of software testing

Software Testing targets a lot of domains. The objectives of Software testing are listed below:

- Finding mistakes which a developer does when he/she develops a code.
- Increasing the quality of the application.
- To make sure that the application is defect free.
- It ensures that the end results are as per the expectation or not.
- To assure the consistency of SRS i.e. System requirement spec doc and BRS i.e. Business Requirement Spec doc.

- In gaining customer's confidence and retaining the customer.

Testing a software helps in comparing the application or product against user and business requirements. It is most important to have good test coverage to test the software application completely and to ensure that it performs well and according to the SRS.

If the test case coverage of the code is high, the test cases have to be very strong with maximum cases of detecting the faults. This objective can be calculated by taking into consideration the count of defects reported per test case. More the number of defects, it means the test cases were made very strong.

Once the code drop is done, and the product is given to the end users or the customers there should be no complaint from them. For making this happen the tester must know as to how the end users are going to use this application and as per that they must design the test scenarios and make the test cases. This will help in fulfilling customer's implicit and explicit requirements.

Software testing makes sure that the product made is ready for a defect free usage. Good coverage implies that the test cases has been made to cover the a lot of areas like compatibility of the software with OS since for different OS a different software can be made, functionality of the application, hardware and various types of web browsers, testing the performance to test the throughput of the software and load testing to check that the system is of good reliability and should not crash or no scenarios in which there is any blocking in the developed application. It also helps determining the correct deployment of the software to the machine and without any issues. This makes the application software easy in installation, learning, and usage.

1.3 Testing Process

Software Testing is a process, not a single action. This process starts from the very first step of SDLC i.e. planning then making the test cases, preparing for the execution of the software and the evaluating the footing till the test closure. So, activities can be divided into the following elementary steps:

i. Planning and Controlling
ii. Analyzing and designing

iii. Implement and Execute
iv. Evaluate exit criteria and make Report
v. Test Conclusion activities

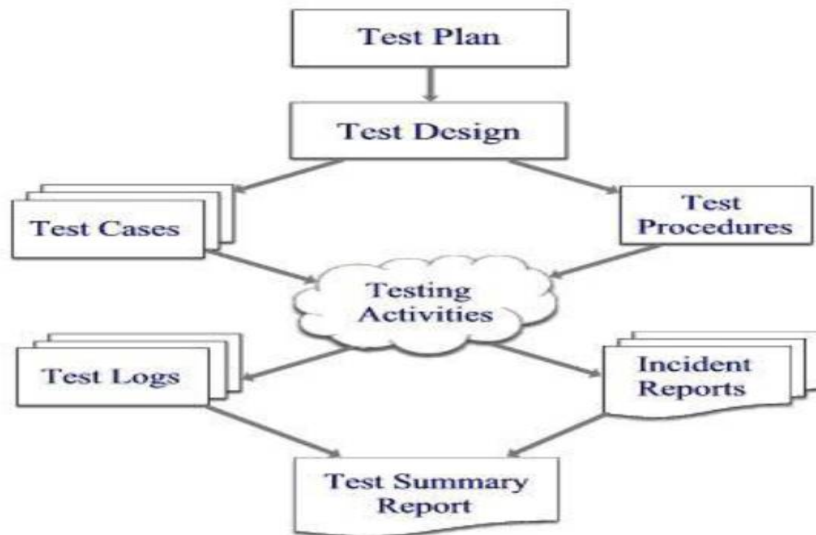


Figure 1.1: Testing Process [3].

1.3.1 Planning and Control:

Test planning has the following things to take care off:

i. Determining the scope, risks and identifying the objectives.
ii. Test approach has to be formulated with the help of test planning.
iii. In implementing the strategy of testing. Test strategy is a concept that gives the description of the testing portion of the SDLC. It is made to inform managers, developers and testers about some key issues of the testing process. This includes the testing objectives, method of testing, total time and resources required for the project and the testing environments).
iv. To determine the required test resources like people, test environments, PCs, etc.
v. To schedule test analysis and design tasks, test implementation, execution, and evaluation.
vi. To determine the Exit criteria we need to set criteria such as **Coverage criteria.** (Coverage criteria are the percentage of statements in the software that must be

executed during testing. This will help us track whether we are competing test activities correctly. They will show us which tasks and checks we must compete for a particular level of testing before we can say that testing is finished.)

Test control has the following major tasks:

i. To measure and analyze the results of reviews and testing.
ii. To monitor and document progress, test coverage, and exit criteria.
iii. To provide information on testing.
iv. To initiate corrective actions.
v. To make decisions.

1.3.2 Analysis and Design:

Test analysis and Test Design has the following major tasks:

i. To review the test basis.
ii. For identifying various test conditions.
iii. In designing the tests
iv. For evaluation of testability of the requirements mentioned in SRS and system.
v. For designing the test environment set-up and in identification and required infrastructure and tools.

1.3.3 Implementation and Execution:

During test implementation and execution, we take the test conditions into test cases and procedures and other test ware such as scripts for automation, the test environment, and any other test infrastructure. (Test cases are a set of conditions under which a tester will determine whether an application is working correctly or not). Test ware is a term for all utilities that serve in combination for testing software like scripts, the test environment and any other test infrastructure for later reuse.)

Test implementation has the following major task:

i. In creating and giving precedence to our test cases by using methods and making test data for those scenarios of test. Some instructions are also written for bringing out the assessment which are known as assessment methods. There is a requirement to automate tests by the use of test harness and by the use of automated tests scripts. (A test harness is

the amassment of software and test data for testing a set of instruction units by running it under several different conditions and detecting its pattern and outputs.

ii. In making test suites from the test cases for well organized and good performance based test execution. (Test suite is the collection of test cases that are used for testing an application program to prove that it has some given set of characters. A test suite often contains in depth information for each collection and category of test cases on the system configuration which is used during the testing phase of SDLC. Test suites basically club similar type of test cases together.)

iii. To implement and verify the environment.

Execution of tests has the following major tasks:

i. In running the individual tests as well as the test suits.
ii. To perform the tests that during the previous situation failed to pass. This is also called as re-testing or confirmation testing.
iii. In logging the generated results of the test execution and record the identities and versions of the software under tests. Audit trial is done for the generated test log. (The basic definition of a test log is the document which contains the in depth details of who created the test, who ran the test, the time stamp, the result i.e. the the test case status (pass/fail)
iv. To In finding the errors generated i.e. the comparison of the actual and expected results.
v. The report of the found out errors is generated and these discrepancies are reported.

1.3.4 Evaluating Exit criteria and Reporting:

On the basis of the risks projection of a particular project the test cases are prepared. If the assessment of risks is high then very strong test cases have to be made. These criteria which vary from one project to another are known as exit criteria.
This appears in the following scenarios:

- Certain ratio is kept for maximum number of test cases.
- The rate of defect needs to be under certain level.
- When deadlines are achieved.

Evaluating exit criteria’s has the following tasks:

i. For checking test logs against exit criteria’s mentioned in the test planning.

ii For assessing that if the number of test cases needed for checking the scenario is more than one or strategy needs to change.

1.3.5 Activities of test Closure:

Upon the delivery of product, test closure activities are performed. There are other reasons for which testing can be closed:

- Upon the cancellation of the project.
- In a situation in which a set target is attained.
- During the maintenance activities of the application.

Activities of test a closure have following things to take care of:

i. For checking if the planned deliverables are actual delivered

ii. For finalizing and archiving test ware for their reuse.

iii. Maintenance organization should be handed over the test ware. This will help in the support activities of the project.

iv For accessing the results of testing and for making a conclusion out of the generated test reports.

1.4 The Testing Spectrum

Every phase of SDLC has testing in it, but the type of testing which is done during each phase of the SDLC is objectively different from other phase. At the minimum level, unit testing is done. The tiniest piece of code is tested i.e. in unit testing each and every instruction has to be tested. When any two or more modules have to be joined or integrated, integration testing is done. This type of testing is done on both the modules and then checked on the combination of the modules.

End to end quality of the application is taken care of by System testing. Both the functional and the non functional specs are checked by system testing. The implicit and the explicit requirements have to be checked during this type of testing

Acceptance testing is done at the end of the customer. It means when the software has been delivered to the client. Thus the testing done at the end of the customer is known as acceptance testing.

1.4.1 Functional Technique and Structural Technique

The step by step process of giving input to the application, checking the results against the received output is testing. Requirements specification, design specification, source code, and so on is all included under software configuration. There are different techniques for analyzing the standard of software developed by the developers.

Functional Testing: The functionality of the software is checked in this part. It is also known as the black box testing. Since the code is not tested or visible in this form of testing, this is known as black box testing.

Structural Testing: The application is viewed as a white box testing. It is because in this form of testing we can view the code available. A lot of rigorous testing is done in this part because every logic is checked for the correct execution of code.

1.5 Software Testing - Methods

1.5.1 Black-Box Testing

The type of testing in which mere functionality of a given application or module is checked is known as black box testing. For a given module if it has to perform certain functionality only the input and the output is supposed to be checked in this part. If the expected and the resultant is same the test cases are passed else a defect is logged in for the same.

Some of the benefits of black box testing are listed below:

- For checking larger application Black box testing is well suited.
- We need not have any hands on code.
- It has a clear view of developer and user's view with the help of visibly defined characters.

- Testers with no prior knowledge of code can test the application.

Some demerits are listed below:

- It offers very limited coverage since the number of test cases for a certain scenario is very less from functional testing perspective.
- Not much efficiency since the quality assurance cell has no in depth knowledge of code.

1.5.2 White-Box Testing

Also known as glass box or open box testing. This is because the code to be tested is available to the quality assurance cell Therefore for performing the white box testing the tester should have in depth knowledge of working of code. Whenever the application is not working as per the expectation the source code of the module can be checked by the tester. The merits of white-box testing are given below:

- It helps in the selection of type of data to be used for testing a scenario. Since we have the in depth knowledge of code.
- Code optimization.
- Maximum coverage of code is achieved in this type of testing.

The demerits of white-box testing are given below:

- The cost of the project increases since the tester has coding skills as well.
- It can be very time consuming since the application developed can a thousand's of KLOC's
- The cost of project increases since debugging tools are used in this part.

1.5.3 Grey-Box Testing

In this type of testing the tester has a very limited known of the code. The code is available for checking. But this access is limited.

The advantages of grey-box testing are as follows:

- Since it is a mixture of both the black box and the white box testing, it has combined benefits of both.

- The testing is done considering end user's perspective or usage.

The disadvantages of grey-box testing are listed below:

- Test coverage is limited.
- There can be duplicity in test case generation .It is because the developer can also make test cases for the same scenario.
- It can time consuming in case every scenario is being considered.

1.6 Mutation Testing

Mutation testing is a type of testing aimed at locating and exposing the weakness in test suites. The main motivation for Mutation testing is to make strong test cases in contrast of finding faults in the source code. Mutation testing aims on strength of test suites which is used for checking the source code it falls under the category of white box testing since the source code is available to us for testing. Mutation testing is also referred as fault based testing. Any small change that differentiates the program from the original program is a mutant. There are various types of mutants: stillborn, trivial, equivalent.

The prerequisite is the source code and a test case for testing that source code. To create a mutant, only thing that is required is to vary the original program by inserting a small fault in it. The mutants are checked by running the original test data. Differences refer the mutant are killed (dead). In case the mutant remains live, the possibility arises either if the mutant and the native program are equivalent or the mutant could not be killed as the test set was inadequate. Traditional mutation testing consists of operators for a mutation that represent syntactically small errors like replacing + by – in an arithmetic expression. There are several drawbacks of traditional mutation testing. Some of the drawbacks are listed: for a small program, the number of mutants produced is large [25], which increases the chances of equivalent mutants. To deal with equivalent mutants, the extra amount of manual work is required. This extra effort increases the cost of testing. Mutation operators do not consider the misunderstandings related to semantic changes; the only concern is syntactic level.

Mutation testing is based on two hypotheses. The first one amongst them is the competent programmer hypothesis. Most of the faults which occur in the application code are due to syntactic errors. This is the agenda on which competent programmer hypothesis is based upon. Coupling effect is the second hypothesis. The coupling effect says that the test data or the test case which is able to detect simple type of bugs are good enough for detection of complex defects. When more than a single change is made in the code we get higher order mutants. Mutation testing is accomplished by first making the operators. These operators are known as mutation operators. Then the test case is given input against the original and the varied code. After that the results are compared for the both of them. If the output comes out to be different from the original one, then the mutant is said to be killed. Mutation testing is done in the following listed steps:

Step l: Some change in the original source code is done by the tester. This produces a varied code or the mutant of the original file.

Step 2: Test suites are executed both to the original and the mutated file.

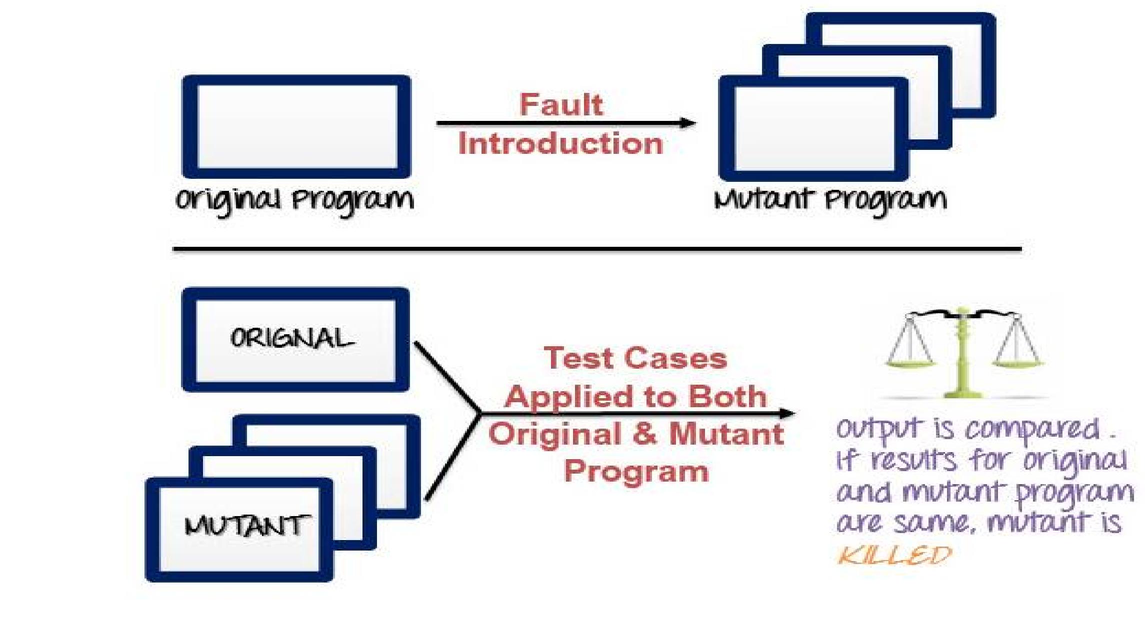


Figure 1.2 : Mutation testing [30].

Step 3: The results which are generated for both the files are checked upon and compared.

Step 4: But in the case when the original and the varied mutated code are the same, it means that the mutant is still alive and the test case has not detected the change. Hence a mutation is just a simple change in the source code. Even if a single change is made to the source code, the file is said to be mutated.

Consider the below code:

```
if (p&&q) {
   r = 20;
} else {
   r = 10;
}
```

We have applied a single mutation to the above code i.e. changed && to ||.

```
if (p || q) {
   r = 200;
} else {
   r = 100;
}
```

If the test has to kill the above scenario, the following set of things should be satisfied:

1. The code must be reachable.
2. The output of the mutated and the original code should be different.
3. The change i.e. (the value of 'r') must appear as output once the testing has been done.

If only the first two conditions are met, the term coined for such a type of testing is weak mutation testing. Since the change is not reflected by the code. So the code is not in a killed state.

On the contrary there are several test cases in which the change can be detected. This comes under the category of strong mutation testing. In this type of testing, all the three conditions are met.

1.7 Mutation Operators

Below are some examples of the mutation operators which are developed during research:

- Deleting Instruction
- Changing Boolean values from false to true and vice versa
- Arithmetic operator are replaced amongst themselves (*, /,+,-)
- Replacing the relational operators(For e.g.> with <, >= with <=)

1.7.1 Mutation Score (MS)

The definition of mutation score is the ratio of Number of mutants which are killed to the total number of mutants.

MS = # of mutants killed / Total mutants

Efficiency of test cases is more when the ratio is close to 1. The efficiency of test suites is calculated with the help of Mutation score. Mutation testing is very expensive due to higher cost of creating mutants.

Advantages of Mutation testing as given below:

- Helps to evaluate the exact functioning of code.
- Helps to make strong test cases which can detect defects.
- Helps to check the loop holes in code.
- Helps to achieve higher quality since it strengthens the test suite.

- Helps to attain higher code coverage.
- More reliable software is made by the help of Mutation testing.

Disadvantages of mutation testing are given below:

- It makes the product expensive due to additional cost of creating mutants.
- The implementation of higher order mutants is tough.
- Testers need to have strong coding knowledge.
- Functional testing cannot be achieved with mutation testing.

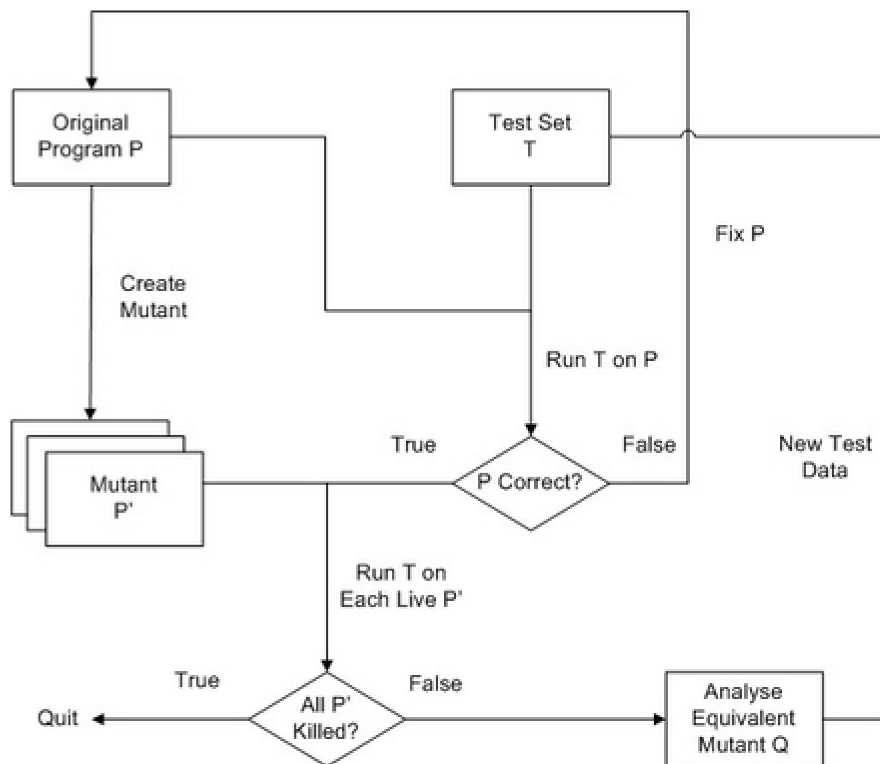


Figure 1.3: Mutation Testing Process [28].

Traditional mutation testing is achieved by making a change in the source with the help of arithmetic operators or by changing the relational operators from one to another type.

1.7.2 Arithmetic Operators

There are five types of arithmetic operators' i.e. +, -, %, /, and *. Since they need two operands for execution, these are binary operators. Since there are operators like ++, -- i.e. and further can be categorized as pre increment, pre decrement, post increment and post decrement operators and these require just a single operand, they come under unary operators.

- Arithmetic Operator Replacement Replace is abbreviated as AORB and it can be applied with binary operator switching..

- Arithmetic Operator Replacement Replace is abbreviated as AORU and they replace the unary operators amongst thes et of unary operators.

- Arithmetic Operator Deletion Delete also abbreviated as AODU are the operators which are used to delete the arithmetic the instruction from source code.

1.7.3 Relational Operators

These operators switch the relational operators between them. The mutated code can have version of any change reflecting <= to >= or >= to <= and so on With the help of Relational operators the value of true in a given expression will change to false when a change is made in the code. Once the change is detected by the test case, the test case will become useful test case.

1.7.4 Conditional Operators

There are five binary conditional operators i.e. 'AND', 'OR','&','|' and '^'. Only one unary conditional operator is there which 'NOT i.e. '!'.

- Conditional Operator Replacement Replace replaces the operators among the five given binary operators. It is abbreviated as COR.

- 'Conditional Operator Insertion' inserts the '!' in the source code. It is abbreviated as COI

- 'Conditional Operator Deletion' deletes the unary operator from the instruction or source program which has '!' in it.

1.7.5 Shift Operators

There are three shift operators: >>, <>, >>. These are bit manipulation operators which shift the bits to either left or right depending on the type of operator used.

• 'Shift Operator Replacement' (SOR) replaces one of the SOR operators with other SOR.

1.7.6Assignment Operators

The basic operators which come under the assignment operators are the ones which help to assign the value from the right hand side of the source code instruction to the left hand side of the instruction. The operators are &=, | =, ˆ=, +=, -=, *=, /=, %=, <<=, >>=, and >>>=.

• 'Short-Cut Assignment Operator Replacement' replaces one of the assignment operators mentioned above with any of the remaining ten operators.

1.8 Semantic mutation testing

To deal with several specific types of mistakes, Semantic Mutation Testing was proposed. The semantics can even be changed by a small change in the syntax. For introducing the semantic mistakes, different ways are available. For SMT-P, change in the syntax of description has been chosen in order to simulate semantic mutation. Three types of MT can be studied under SMT-P i.e. weak MT, Firm MT and strong MT.

In case of strong mutation testing, the program, and the mutant can be separately identified, if they produce different outputs for a same test case. On the other hand, in weak mutation testing, if the program and the mutant reflect a value which is not same for a variable immediately after the particular point at which the program was mutated are said to be distinguished.

With the help of Firm MT we can in genral allow the quality tester or the debugger to select the position at which the value of a variable can be changed. In SMT the semantics of that particular language is denoted with the symbol 'L', and in totality the behavior is described by the use of pair i.e. (N,L). If there is any change in the traditional form of MT then the description of mutant will be changed to (N', L). But in case we alter the the semantics of the given language the description would be given as (N,L').

1.9 Thesis Outline

This thesis is organized into five chapters. Chapter 1 describes the basic concepts of Software Testing and Mutation Testing. It also consists a brief introduction of the types, conditions, examples and operators of mutation testing. Chapter2 describes literature survey that has been done during this research. Chapter3 describes the motivation behind the dissertation, discusses the problem statement and its objectives. Chapter4 explains all the results obtain from the algorithm that has been developed for reducing the total time for executing mutants. Chapter5 summarizes the conclusions drawn from the work done.

Baldwin et al. [11] proposed a compiler optimization technique for the equivalent mutant detection. The procedure used for optimization of the program will make an equivalent application, so the detection of the equivalent mutant is possible if a reverse process is followed. This is a basis of the approach.

Bousquet et al. [12] proposed that there can be an evidence for the source code, since luster is based on mathematical foundation. The authors used lESAR [l3, l4], a model-checker for luster that can be used prove the correctness of an application or to compare two programs. It needs to compare a mutated and an original source code. In case the description of the environment is provided, mutant equivalency can be taken into consideration.

Ellims[15] describes in the his research work that initial results from the research on mutation tool for C language. Suggestions for future research are the most valuable part of this work from the perspective of equivalent mutant's problem: "Firstly to observe at possibilities for changing the source code to prevent difficult to- kill variations being generated. On the other hand looking at other effects of mutants which cannot be side lined such as Usage of CPU, usage of memory, etc. as a means of knowing about the non-equivalent mutant.

M. Harman et al [l7] have given three ways in which a given variation of original code can be categorized: strong, weak and firm. An approach proposed in this paper uses firm mutation testing [l8]. The authors assume that "mutants which fail to propagate 'corrupted data' to the inspection set at the probe point will be equivalent and should be avoided"

Y. Jia [l9] and M. Harman [l9] have defined the concept of newer paradigm for Mutation Testing, in which the order of mutation has been increased. When we consider traditional MT only the single order mutation scenarios are considered. But if more than one change

is made in the original source code, the concept is known as higher order MT (HOM). The HOM helps to reduce costs, reduce the count of generated equivalent mutatnts

G. Kaminski et al. [20] suggest weak mutation testing (state-based inspection) and a collection of logic operators only allowed the authors to introduce a fault class hierarchy. In this hierarchy, they have selected operators that do not create equivalent mutants

K. Fisler et al. [22] explained a change-impact analysis tool was used to find out number of equivalent mutants in the total number of mutants[23]. The authors originally had an original belief that "detection of equivalent mutatnts is an important efficiency improvement concept though in reality they found that the evaluation of request to be computationally cheaper than doing the analysis of change impact with Margrave. Furthermore, there were a lot of limitations on the detection of equivalent mutants for mutation operators on conditions.

J. Offutt et al. [24] have introduced specific methods which help for Java class mutation operators that are from constraint solving techniques. Instead of going in a `post-processing mode', after generation of mutants, MuJava has integrated the equivalent mutation analysis with the mutant generation, a concept given by Harman et al. [24]. MuJava helps to implement channelized heuristics that avoid the equivalent mutants for specific mutation operators. This method is based on equivalency conditions of mutation operators, which in turn is the basis of the possibilities under which mutants are killed [24]. The authors and researchers have given equivalency conditions for sixteen mutation class-level operators.

D. Schuler [26] and A. Zeller [26] have given description of equivalent mutants and observed the impact on the applications output. This impact of a variation can be analyzed by cross checking the code state at the end of the execution, as tests do. The assessment of the impact of a variation while the execution is being performed can also be done. In particular, they can analyze the variation in the source code behavior between the mutated and the original version of the code. The concept is that if a mutated code is impacted in its internal code behavior, it is likely to change the external source code behavior as well and therefore impacts the meaning of the program. If they channelize on

mutated versions with impact, they will therefore expect to know that there are fewer equivalent mutants [26].

In [27], Meimandi et al. has made their contribution to the field of aspect oriented software testing. They have covered issues faced, bug models and their types, testing coverage, various testing methodologies, automated support and maintenance, and conducted various empirical studies. An entire field of AOP testing has been shown in this paper and this can help the researcher and students, who are relatively new, in gathering knowledge on this concept and also given further avenues of exploring for motivated researchers[28].

In the modern times there is an option of using the open source tools which are available. This has led to the concept that mutation testing in the newer times has reached a mature state. In the earlier application perspective of mutation testing only the simple mutations were performed but now a days the focus is increasingly shifting towards elaborate forms of mutation testing. Rather than channelizing only the concept of traditional mutation testing the researchers are shifting towards the concept of semantic mutation testing also abbreviated as SMT. This interest towards SMT has led to the increased research work towards higher order mutation testing [29].

The concept of semantic mutation testing has been given a conceptual new definition in [30]. Their focus is to present potential misunderstandings of the meaning of a particular description language. A range of situations has been described in which semantic mutation testing can play a very important role and also have given description about a semantic mutation testing tool that has been developed. Traditional mutation testing mutates the syntax i.e. basically the source code of a description N to generate some mutant N′ . The mutant is in general created by the application use of a mutation operator. If there is a difference between N and N’, the test case is said to be killed.

A test suite is efficient and useful if it detects the change in every (non-equivalent) formed mutant. The concept is that the mutants can simulate potential errors and thus that a test scenario that detects the non-equivalent variants will be efficient at finding such errors. However, the pattern which is in link with a description is defined by a set of the syntax N of the description language and the semantics L of the description language in

which it's description is given. Thus, traditional mutation operators provide a transition from form (N, L) to the form (N′, L). In SMT the meaning i.e. basic constructs of the description language L are varied, out of which mutants are created. Thus, the application of a SMT operator is of the form (N, L) → (N, L′). Semantic mutation testing has a focus to simulate the errors that are a due to reasons misunderstanding of the semantics or the basic constructs of the description language used. It is although argued that SMT captures a different domain type of mistakes in comparison to traditional mutation testing.

Let us consider a situation in which a firm has changed from one tool to another tool, and thus from one semantics to another. Here there are some possibilities that mistakes may occur since the semantics followed in the previous language were different from the one which will be used now. SMT can be considered to take a check on these kinds of issues and so the process of migration from one kind of tool set to other could be taken care of by semantic mutation operators set.

Haitao et al. [31] discusses his experiences in regard to the research work and development of a SMT tool which is based in C language: SMT-C. In addition to provision of basic things, they had set target on achieving the following objectives: Weak MT/SMT in C; excellent portability; embedded in to the routines of a C developer; an easy to use front-end. Though SMT-C is still under development, with the research work and experience they have, it is argued that the method adopted in SMT-C is a handy method for these objectives. For creating semantic mutants, they have used a tool named TXL [31] which changes the code from one form to another. The TXL method has no shortcomings on platform, specific language and compiler. This has ensured that the creation of mutant in SMT-C is highly portable. Also it applies to many other programming languages. This is very important as there is plan to check SMT against different abstract descriptions.

A. J.Offutt [32] presented a lot of conclusions about the coupling effect as calculated over the domain of mutation analysis. At the first step, they found out that the test data which is created to kill single order variants are very successful at killing second order mutants. Of course, there is the fact that two mutants are executed on the same path does not necessarily mean that they interact in any meaningful manner, but such interactions are

quite hard to determine with mere analysis. By including all second order mutations on the same paths, they have included all those than can have interaction. Second, they observed that the second order mutants that were not killed show no marks that would create confusion in mind that it was very tough to kill by using test data developed for single order mutants. Third, they have founded that the test data set made for single order mutants had killed a relatively more percentage of mutants when applied to second order mutants. Finally, test data generated for l-order mutants killed a higher number of mutants when checked against to third order mutants

MESSI [34] (Mutant Evaluation by Static Semantic Interpretation) was given to find out mutants which are tough to kill. It achieves this concept by comparison of their symbolic output with static analysis. MESSI does not always selects the most tedious to kill variants. Though removal of half of the mutants has a very crucial effect on the mutation score and probability of killing other mutant, there is the small difference between the twenty-five and fifty percent samples. In case of TCAS, MESSI has small effect on the mutation score. Therefore, MESSI is a useful technique, but the efforts must be done to increase its reliability. The present version of MESSI evaluates semantic difference without taking path conditions into consideration. Mutated path conditions can leave the symbolic output unchanged.

Offutt et al. [37] have extended their six selective mutation operators further using a similar selection methodology. Based on Mothra[37] they have categorized the statements in three parts i.e. operands, statements and expressions. They have tried to delete operators from every class in turn. They found out that five operators from the list of operands and expressions class had become the main operators and the five operators are lCR, ABS, UOI, AOR and ROR. These main operators calculated a mutation score of 99.5%.

Offutt and lee [42, 43] has presented a comprehensive empirical study using a weak mutation system named leonardo. In this research work, they used the 22 mutation operators in Mothra as fault models instead of Howden's set of five. The findings from their research work have shown that Weak Mutation is an alternative to Strong Mutation in most of the common cases, agreeing with the probabilistic findings of Horgan and

Mathur and research work results of Girgis, Woodward and Marick. The most recent work on the weak mutation was conducted by Durelli et al. [44].

They did extension of Java virtual machine to perform analysis of weak mutation. Research work results prove that the virtual machine based developement achieves has speeded up more than 80% in comparison to traditional mutation.

Chapter 3 Problem Statement

Compilation done separately is slower than byte code translation. The speed can be improved. However, the execution of mutants using these approaches takes more time. Many tools have been developed for mutation testing. Some of the examples are MuJava, JavaMut, Proteum, SMT-C for C language. No such tool is available which fulfills the requirements of SMT. By developing an effective algorithm we can provide less time for the execution of mutants.

3.1 Dissertation Objectives

Following are the research objectives based upon on literature survey.

- Analysis of semantic mutation testing technique.
- To generate the semantic mutation operators for python.
- To propose a semantic mutation testing tool for python.
- Comparison of the semantic mutation testing and traditional mutation testing

3.2 Experimental Setup

The following tool has been used to perform this study:

3.2.l PYCharm IDE

In programming, Pycharm is an IDE i.e. integrated development environment. For customizing the environment it has plug in system and a base work space. First grade support for Python, JavaScript, Coffee Script, Type Script and CSS is given by PyCharm's smart code editor. Benefit of language aware code completion and error detection can taken. For modern web development frame works such as Google App Engine, web2py, Django, Pyramid, and Flask, PyCharm offers great frame work specific support. PyCharm can work on either of the operating systems like Windows, linux or Mac even with a single license key. Enjoy a fine-tuned workspace with customizable color schemes and key-bindings, with VIM emulation available.

3.2.2 Python

Python is a one of the most widely used general-purpose, high-level programming language. It is designed in such a manner that it supports the readability of code, and its format syntax helps developers to code the project particulars in lesser number lines of code(LOC) in comparison to other programming language such as C. The language has provided constructs which intend to make clear pieces of code on both a small and a large scale.

Python aids multiple programming paradigms, which include functional programming or procedural styles, imperative, object-oriented. It has a dynamic type of system, has a very big and comprehensive standard library and automatic memory management.

In similarity to other dynamic languages, Python is used as a scripting language but is as well as in a wide field of non scripting contexts. Usage of third-party tools, such as Pyinstaller, code of Python can be contained into stand alone executable(.exe) source code programs. Interpreters of Python are available for different operating systems (OS).

CPython, is a reference development of Python, is open source and free software and has a development model which is community based,. The non-profit Python Software Foundation manages CPython .

3.2.3 Unit Test: PyUnit is the unit testing framework in Python language. It is version of JUnit, by Erich Gamma and Kent Beck. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. For respective language each one is the de facto standard Unit testing frame work.

The unit test has support for shutdown code for tests independence of the tests from the reporting framework, sharing of setup, aggregation of tests into collections and test automation. The unit test module has provision for classes which make it simpler to give support for these qualities for a test set.

In order to achieve this, units test has some very important basics:

Test fixture A test fixture presents the preparation which is needed to do one or more than one tests, and any kinf of associate clean up actions. This may have involvement of, for instance, creation of proxy or temporary databases and directories.

Test case The smallest unit of testing is test case. For a particular set of inputs it looks for a specific output. Base class is provided by the Unit class. The name of the base class is Test Case, which is used making new test cases.

Test suite It contains basically the test aggregation of test cases individually or test suites which would be executed like a batch.

Test runner A test runner is the basic component which arranges for the execution of tests and gives the result to the user. Graphical or textual interface may be used by the runner and can return values which indicate the result status of the executing tests.

The test fixture and the test case basics are supported with the help of the FunctionTestCase and TestCase classes. FunctionTestCase should be used in case of creating new tests, and Test Case can be used while integration of the existing test code with a unit test driven framework. When the building of test fixtures is done, the tearDown() and setUp() methods are used.

These methods can be overridden for providing cleanup and initialization for the fixture. With the help of FunctionTestCase class, the existing functions can be passed as arguments to the constructor of the program for these purposes. When the test is executed, fixture initialization is run at the first step. If the execution is successful, then like a pipeline cleanup method is executed, without worrying about of the output of the test. Every instance of the TestCase is used to run a only one test method, so a new fixture has to be made for every test.

Test suites are developed by the class named TestSuite. The aggregation of individual test or test suite can be done with the help of Test Suites class. For execution of the test suites each test is directly added to the test suite.

Chapter 4 Implementation

4.1 Working of Semantic Mutation Testing

We need to be very sure about the credibility of the unit tests. How can we make sure if the unit tests created are good enough? If the unit tests unable to identify the bugs in the application, it does not mean that there are no defects in the application being tested. May be there are some scenarios which are not caught.

One of the ways to make sure that the test cases developed by the tester are good enough is Mutation testing. It involves involves deliberately alteration of a source code of application software, then re-executing a suite of unit tests against the changed source program. If the unit test is good enough, it will detect the change made to the original code; if not the test case would fail.

Mutation testing is done in the given way. At the first place one needs a source code for testing which has all the test cases made and which pass. Once you have the set of test cases, you run it against the mutated code which was created by making an alteration the original program.

The extent of mutation decides the order of mutation testing. If only one change is made to the given code the mutation testing is said to be single order mutation testing. If more than one change is done and tested, then the type of testing is known as higher order mutation testing.

Now the output of both the cases is duly noted. That is the output of the test cases against original code and then the mutated i.e. the changed code. Then the comparison of both the outputs is done and the efficiency of the cases is done based on results.

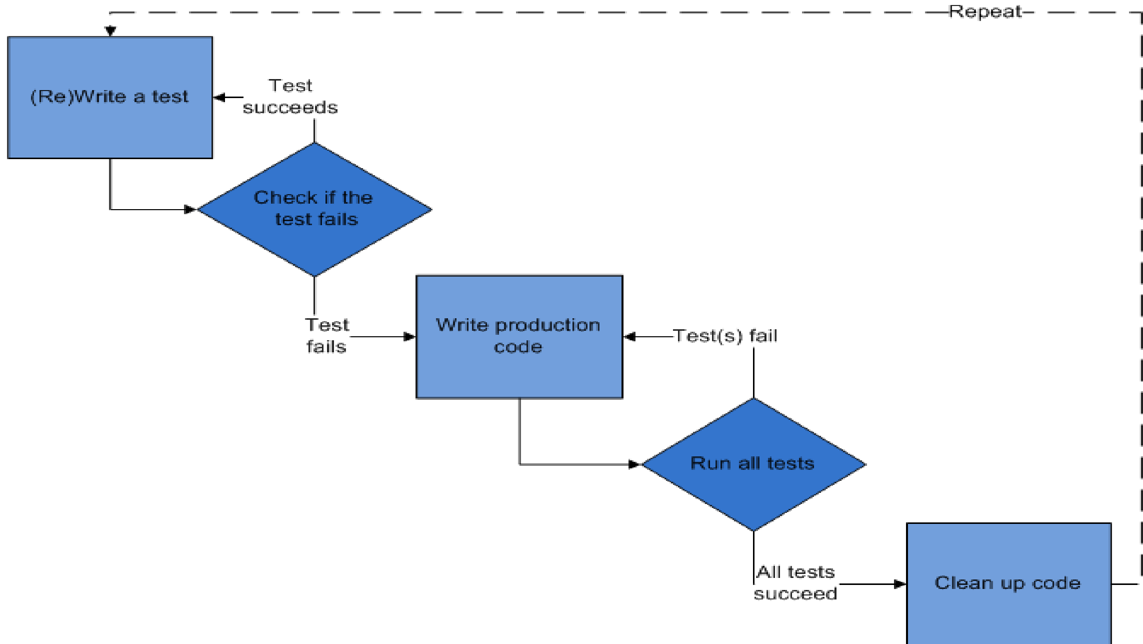


Figure 4. 1 : Working Of Semantic Mutation Testing [37].

4.2 Proposed Technique

In the proposed technique, a tool SMT - P for semantic mutation testing is developed using Python that calculates the efficiency of the test suite. In this, the changes made in the syntax of the description reflect the simulations in the semantic mutations. The aim was to design a flexible and an easy to use tool. In the following part, the semantic mutation operators are described. We consider the programs to be written in a language l and description N. The behavior is defined by the combination of description and language i.e., (N, L). In traditional mutation testing, the mutant of the above description can be (N', L) as there is a change in the syntax. But for semantic mutation testing the description can be represented as (N, L'). For any source code Q, mutant of Q is created according to the single changes done to some instruction in Q. Thus, every mutant of Q will differ from the original source code by only one instruction. The pattern in which the instructions are modified is dependant on the collection of mutation operators applied. For instance, the operator AORB (arithmetic operator for the replacement of binary operator) will change every presence of an arithmetic operator by each of the available possible arithmetic operators. If in program there is a statement Q+P then Q +P yields the following four changed statements: Result = Q - P; Result = Q * P; Result = Q /P; Result

= Q% P. Thus, other operators like unary, logical, relational,, operators can also be used like this. In case of Semantic mutation testing operators, like ASD are used. In ASD: delete all extra semicolons after the condition expressions that is the if statements; It is possible that some developers punctuate if statement as follows: if (a == b);{...}. In this case, the instructions in the bracket after the semicolon will always get executed.

4.2. 1 Proposed Algorithm

The work is done in five steps. Figure 4.2 explains the steps followed in carrying out the research work. A step by step procedure is being followed to conduct the work. To calculate the efficiency of test suites, we first consider a set of standard programs that are used in the project to produce mutant. Then we feed a set of programs to the mutant generator tool that produces thee set of mutants. These mutants are developed according to traditional mutant operators and mutant semantic operators that are specified in mutant generator tool. Then in the third step test suits are applied on the set of mutants to check whether mutants pass or fail. Then we have calculated the efficiency of different test suites for different operators.

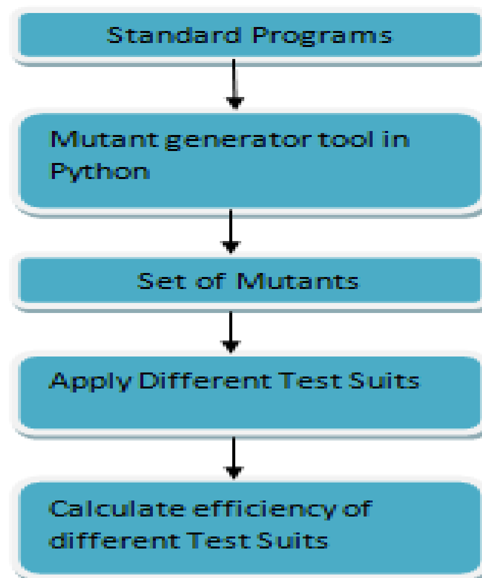


Figure 4.2 : Steps of Proposed Work.

4.3 System Architecture and Implementation

We have made a mutation testing tool based in Python, called SMT-P. The motivation for the development of SMT-P came from the fact that there is no such tool which satisfies the basic requirements of SMT. In this, the changes made in the syntax of the description reflect the simulations in the semantic mutations. The focus was to design an flexible and user friendly tool. The function components have been designed to build, test and execute mutants.

Step l: Choose a set of Standard programs

To perform testing, the basic requirement is the presence of programs on which testing is to be performed. Here we took a number of standard programs for performing the experiments. These programs were converted to Python language. The list of programs used is shown in the table 4.l below:

Table 4.1 : Set of Programs

S.No.	Program Name
L	TCAS
2	Subroutine Find
3	Triangle
4	Trim
5	Check Word
6	Subroutine Max
7	Weeks
8	Calc

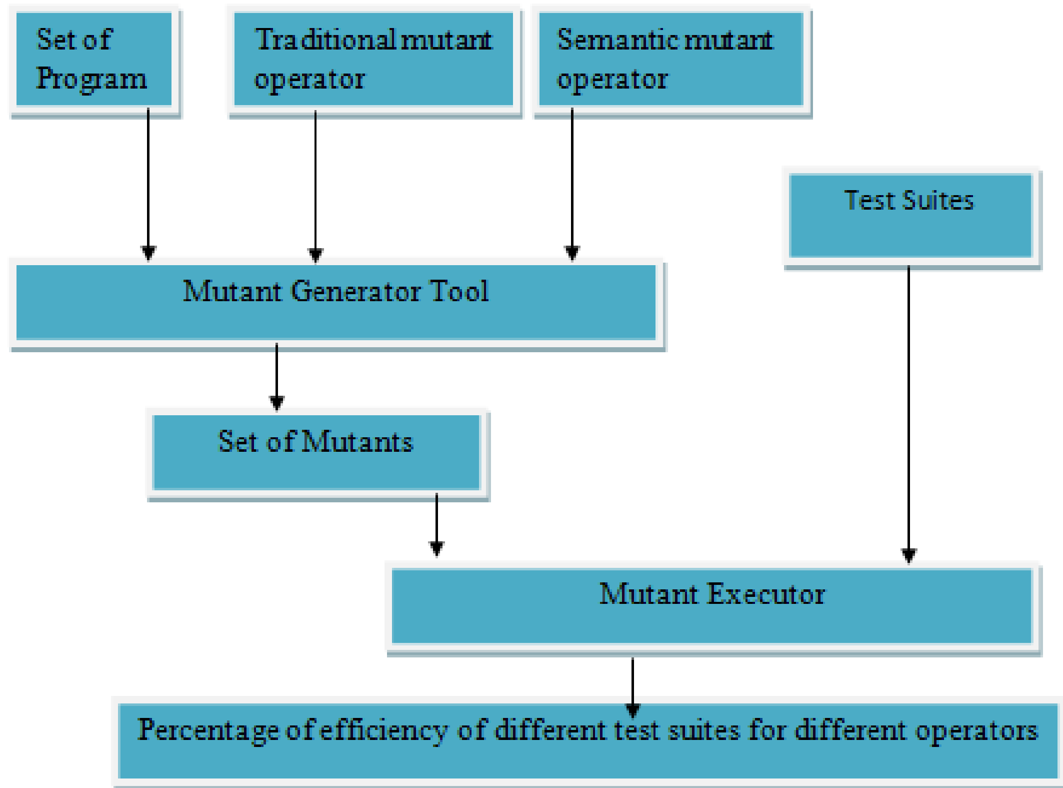


Figure 4.3 : System Architecture

Step 2: Traditional mutation Operator and Semantic Mutation Operators

In semantic mutation testing, the semantics of the description language in which the source code is written according to semantics 'L', the behavior is defined with the pair (N, L). In traditional mutation testing, the variant of the above description language can be (N', L) since there is a change only in the syntax of the source code. But for semantic mutation testing the description can be represented as (N, L').

Traditional mutation testing has operators for a mutation that represent syntactically small errors like replacing + by – in an arithmetic expression.

We have also implemented some of traditional mutation operators in python. The list of traditional mutation operators I showed in the table 4.2 below:

Table 4.2: Traditional Mutant

S.No.	Mutant
L	Break To Continue
2	Continue To Break
3	Arithmetic
4	Statement deletion

We have implemented some of the semantic mutation operators in python. The list of semantic mutation operators is shown in the table 4.3 below:

Table 4.3 : Semantic Mutants

S.No.	Mutant
L	If-Else
2	last case of switch
3	Default for switch
4	Floor of Division
5	Ceil of Division
6	Indentation
7	Elif

1. If – Else: To those 'if' constructs which do not have an else branch, an else branch is added.
2. The last case of switch: When using a switch case without a default branch, a default branch is added.

3. Default for the switch: Here the last branch of the switch statement is modified to be the default statement of the switch.

   The above two operators are designed to ensure complete branching structures. Incomplete branches can lead to errors, as the programmers could assume the execution of last branch o structure.
4. Floor of division: Truncating the float value using the floor method.
5. Ceil of division: On division operations, the quotient depends on the methodology defined i.e. the result can either be the just the immediate previous integer of the quotient or the immediate next integer to the quotient. Here the tail method is used to form the mutant.
6. Indentation: Indentation plays a major role in python. As there are no starting and closing brackets. Here the indentation of statements is changed to see the outcome of test cases.
7. Elif: Wherever in an 'if' statement followed by an 'else- if' statement, else is missing, an else statement is added.

Step 3: Choosing the Test Cases

A test case is considered to be efficient if it is able to kill all the variants created from its source code. We have implemented certain test cases on both the original source code and the mutated code for checking if they produce different results. If the result is different, then the test case passes as if it able to detect the change in the program.

Step 4: Mutation Generator Tool

Mutation Generator Tool has been implemented in Python. The working of SMT-P has been described in the steps below:

Step l: When we run the mutant generator, it first asks to select the source program for which we want to generate the mutants.

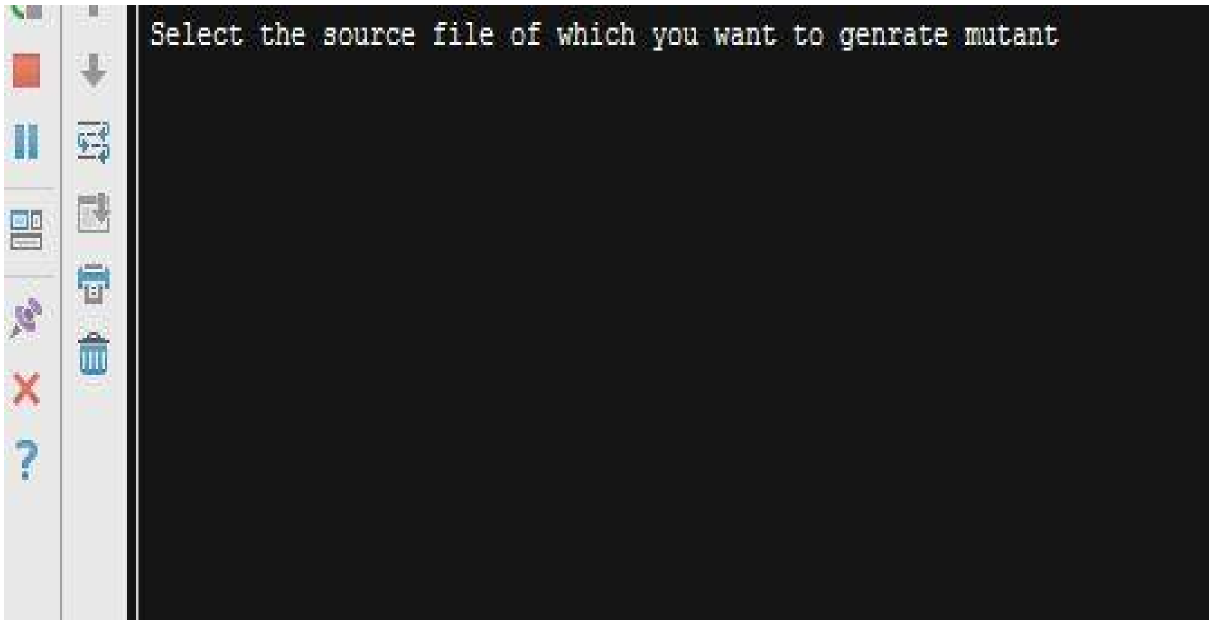


Figure 4.4 : Selecting the source program

The list of available programs is displayed. From this list, we can choose the program for which we want to generate the mutants.

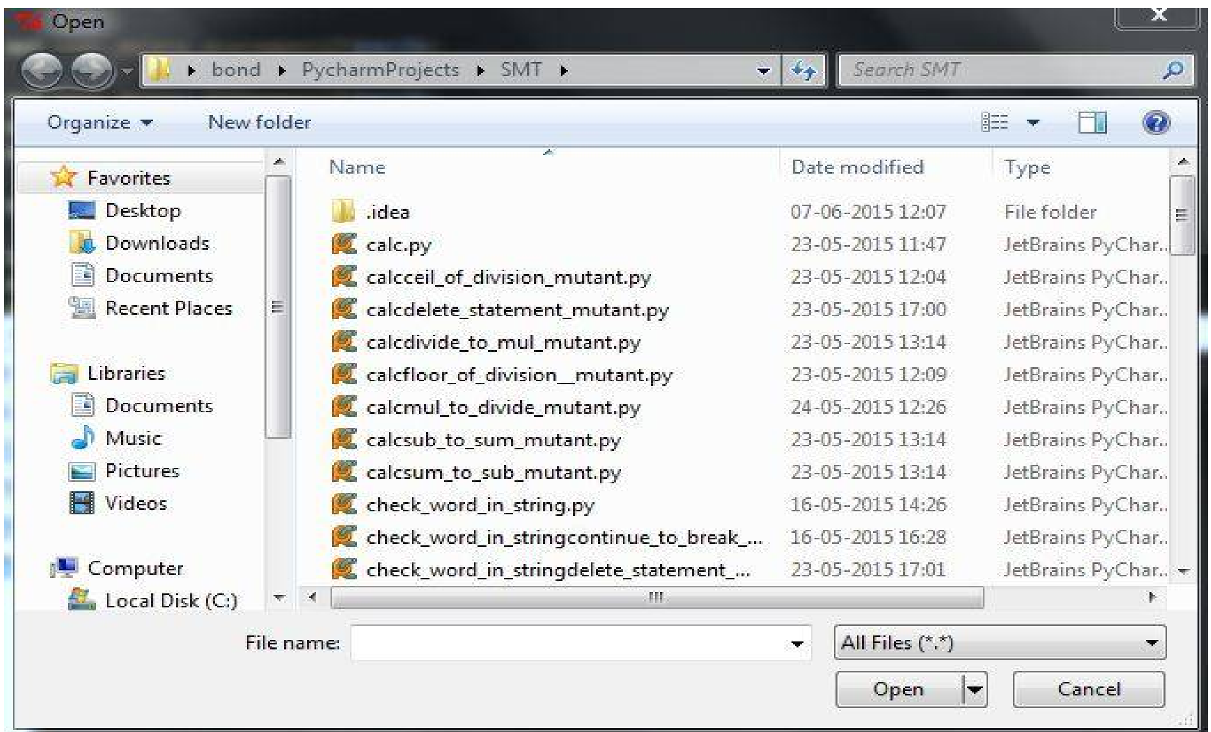


Figure 4.5 :List of available programs

Step 5: Set of Mutants

After selecting the source program, it gives a choice of mutants that can be applied to that program.

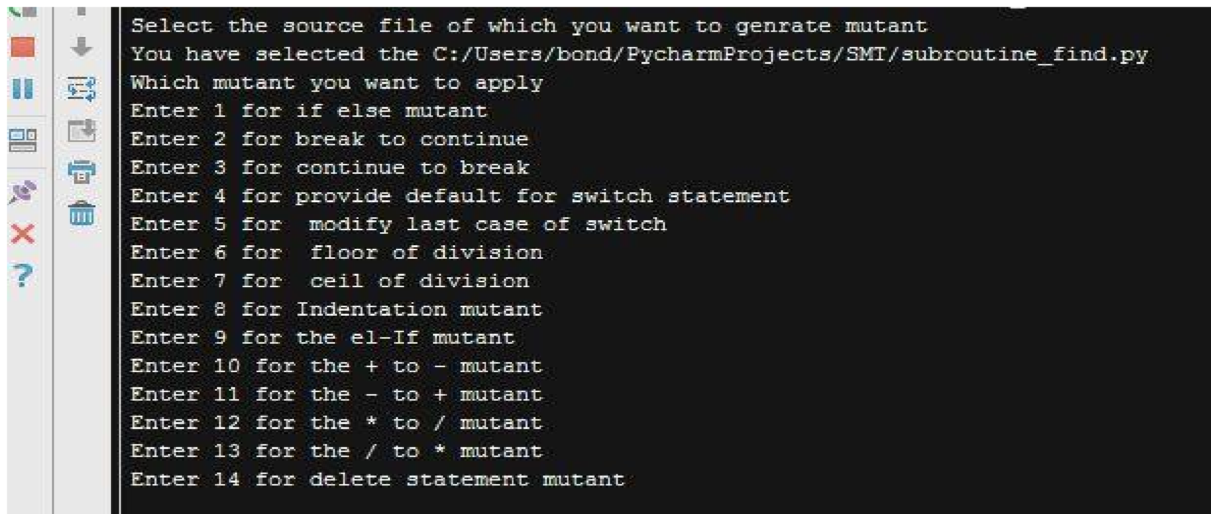


Figure 4.6 : Possible mutants to be applied to the selected source program

From the available list of mutants, one of the mutants is chosen and is applied to the selected source program.

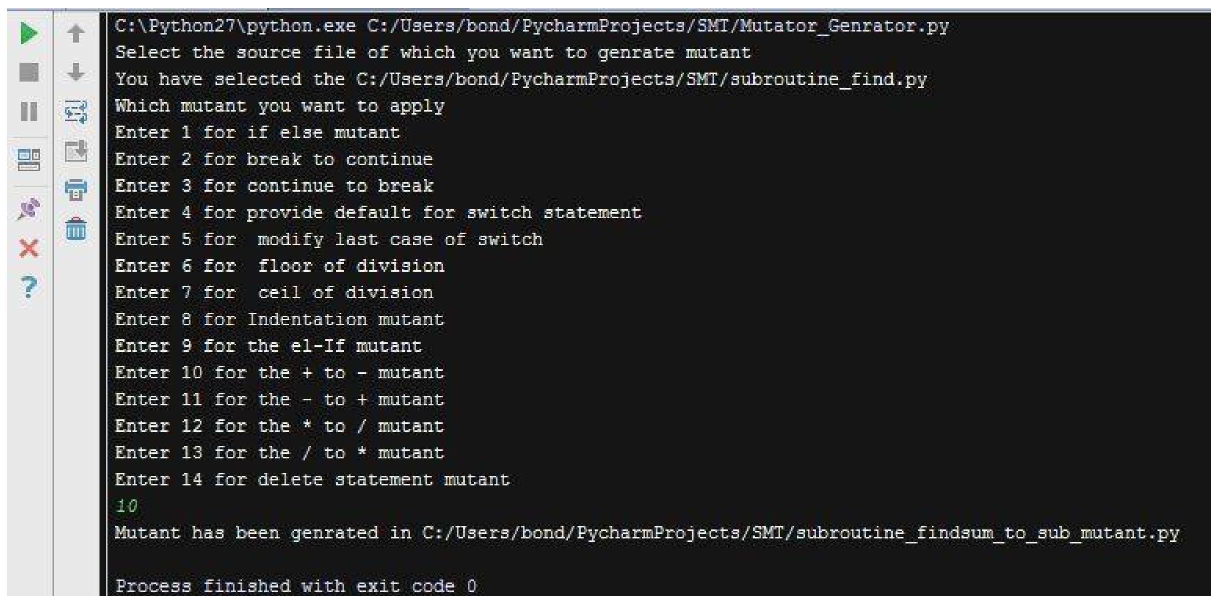


Figure 4.7 : Selecting the mutant

By applying the mutant operator, a mutant of the program is formed. Here we have selected the sum to sub mutant operator. After applying the program, the result is shown below. The original program is shown in figure 4.8 and the mutant generated by applying sum to sub operator is shown in figure 4.9.

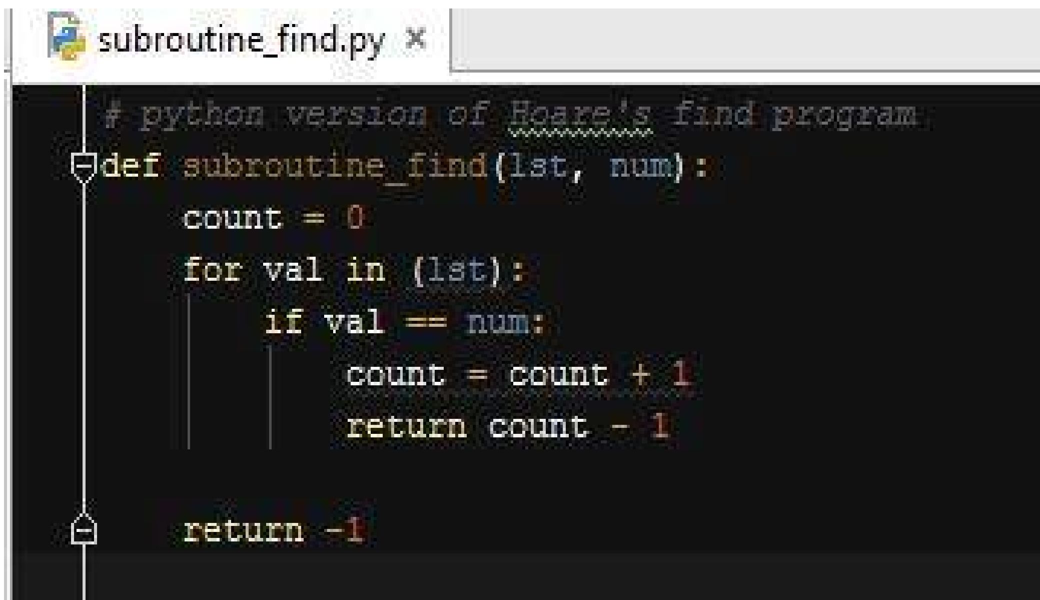


Figure 4.8 : Original Program

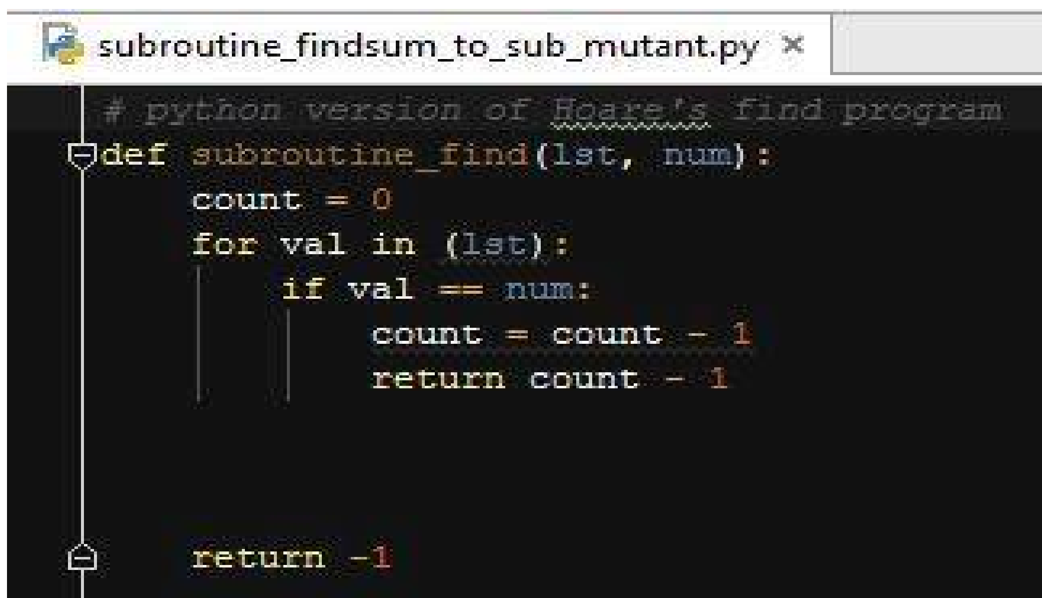


Figure 4.9 : Mutant Program

Step 6: Mutant Executor

PyUnit is the unit testing framework in Python language. It is version of JUnit, by Erich Gamma and Kent Beck. JUnit is, in turn, a Java version of Kent's Smalltalk testing framework. For respective language each one is the de facto standard Unit testing frame work.

Mutant executor runs test cases. A test case that gives different results when run for the actual program and its variant is said to be failed. If a test case does not pass, it means that the test case strength is high and the mutant is said to be killed.

In figure 4.10 the test cases that produce different output with mutant and the actual program have been shown. These test cases kill the mutants.

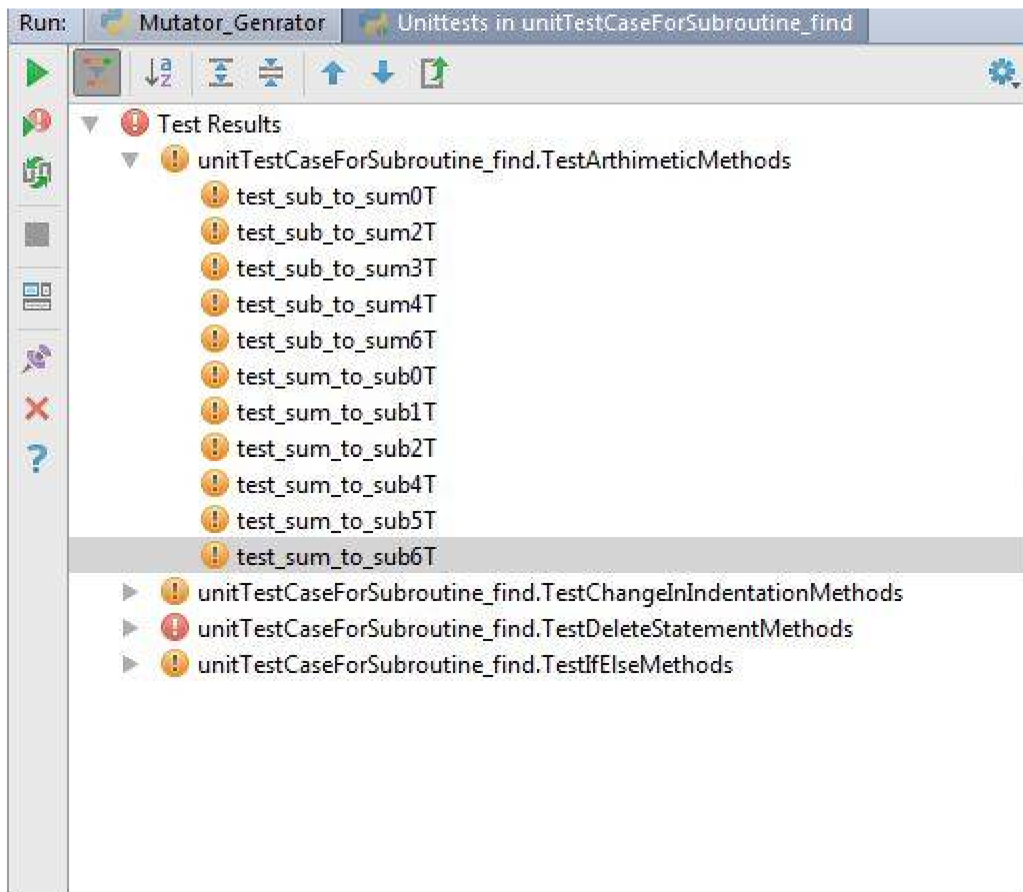


Figure 4.10 : Output in Mutant Executor- Failed Test Cases

In figure 4.11, the reason for failure of a test case is given. If we consider the first case in the figure, we see that there is a failure of the test case. The method called was equal, but the result of the mutant and the original source code is not equal. So the test case failed.

Done: 8 of 42 Failed: 34 (0.342 s)

```
C:\Python27\python.exe "C:\Program Files\JetBrains\PyCharm Community Edition 4.5.1\helpers\pycharm\utrunner.py" C:\Users\bond\Py
Testing started at 00:37 ...

Failure
Traceback (most recent call last):
  File "C:\Users\bond\PycharmProjects\SMT\unitTestCaseForSubroutine_find.py", line 179, in test_sub_to_sum0T
    self.assertEqual(1,subroutine_findsub_to_sum_mutant.subroutine_find([2,12,27],12))
AssertionError: 1 != 2


Failure
Traceback (most recent call last):
  File "C:\Users\bond\PycharmProjects\SMT\unitTestCaseForSubroutine_find.py", line 188, in test_sub_to_sum2T
    self.assertEqual(5,subroutine_findsub_to_sum_mutant.subroutine_find([54,8756,24,47,27,73,45],73))
AssertionError: 5 != 2


Failure
Traceback (most recent call last):
  File "C:\Users\bond\PycharmProjects\SMT\unitTestCaseForSubroutine_find.py", line 193, in test_sub_to_sum3T
    self.assertEqual(-1,subroutine_findsub_to_sum_mutant.subroutine_find([54,8756,24,47,27,73,45],403))
AssertionError: -1 != 1
```

Figure 4.11 : Output of Mutant Executor – Detailed View of Unit Test

The output window in figure 4.12 shows the number of test cases that have passed, failed or throw an error. P here stands for the passed test cases; F stands for failed test cases, and E stands for error that leads to failed test cases.

Test	Time elapsed	Results
unitTestCaseForSubroutine_find.TestArthimeticMethods	0 s	F:11 P:2
unitTestCaseForSubroutine_find.TestChangeInIndentationMethods	1 ms	F:9 P:2
unitTestCaseForSubroutine_find.TestDeleteStatementMethods	<UNKNOWN>	E:6
unitTestCaseForSubroutine_find.TestIfElseMethods	0 s	F:8 P:4

Figure 4.12 : Output of Mutant Executor: Summary of Test Cases Result

4.4 Results and Discussions

From table l and 2, we see that we have applied seven semantic mutation operators on a few programs to produce mutant o each program. Then particular test suites have been applied to the source code and the mutant to check the efficiency of the test suites.

Table 4. 1 : Efficiency of test suites in MST

Program	If-else	Last case of switch	Indentation	Floor of division	Default for switch	Ceil of division	Elif	Total	Failed	Pass	Result
Tcas	10							10	7	3	70.00%
Trim	11							11	7	4	63.64%
Triangle			10				12	22	16	6	72.73%
Subroutine Max	11		11					22	14	8	63.64%
Weeks		11			12			23	4	19	17.39%
Calc				12		12		24	14	10	58.33%
Check Word	11							11	7	4	63.64%
Subroutine Find	12		11					23	17	6	73.91%
Total	55	11	32	12	12	12	12	146	86	60	

Table 4.2 : Efficiency o f test suites in Traditional Mutation testing.

Program	Break To Continue	Statement Deletion	Continue To Break	Arithmetic	Total	Failed	Pass	Result
Tcas		4			4	1	3	25.00%
Subroutine Find		6		13	19	17	2	89.47%
Triangle		8			8	5	3	62.50%
Trim	17	15			32	26	6	81.25%
Check Word		6	13		19	15	4	78.95%
Subroutine Max		8			8	8	0	100.00%
Weeks		9			9	8	1	88.89%
Calc		10		31	41	35	6	85.37%
Total	17	66	13	44	140	115	25	

For eg In Tcas, for the mutant generated by 'if - else' operator a total of l0 test cases have been applied out of which seven test cases are able to distinguish the mutant from original program. As a result, we get 70 % efficiency of the test suite. Similarly for other programs also, the efficiency, has been checked upon. Test suite for subroutine Max is l00% efficient in case of traditional mutation testing, for Trim it is 63.64% efficient in case of semantic mutation testing. To calculate the efficiency (killed/total)*l00 is

calculated. Similarly for both semantic and traditional testing mutation operators are applied and the efficiency has been calculated. We have calculated that semantic mutants are harder to kill than traditional mutants.

We found that test suites that kill all of the traditional mutants also killed approximately 63% of the semantic mutants. However in Tcas, we have found semantic mutant kills 70% and traditional mutant kills 25%.We also found that 58% of test cases are failed by semantic mutant and 82% of test cases are failed by the traditional mutant. The fact that neither set subsumed the other suggests that ideally both semantic and syntactic mutants should be used. Most of the semantic mutation operators are not more complex to apply in comparison to the traditional mutation operators.

The traditional operators just deal with the syntax, and the semantic mutants deal with semantics. So by using both of these operators we see that our tool SMT-P successfully kills both types of mutants.

Chapter 5 Conclusion and Future scope

5.1 Conclusion

One of the effective techniques for testing is mutation testing. Mutant can be created by changing the syntax of a program. To distinguish the mutant from the original program, an effective test suite is required.

We have introduced a new tool SMT-P based on semantic mutation testing. In semantic mutation, the language is modified to produce the mutant. There can be misunderstandings in regard to the semantics of the description language. When the syntax of a description is mutated, it is traditional mutation testing. On the other hand, when we deal with language, it is semantic mutation testing. A test case that produces different results when run on the actual program and its mutant is said to be failed. When a test case fails, the mutant is said to be killed. Here we have calculated the efficiency of different test suites for different operators. We have investigated the efficiency of test cases using SMT-P on both traditional and semantic mutation operators.

We found that test suites that kill all of the traditional mutants also killed approximately 63% of the semantic mutants. However in Tcas, we have found semantic mutant kills 70% and traditional mutant kills 25%.We also found that 58% of test cases are failed by semantic mutant and 82% of test cases are failed by the traditional mutant. The fact that neither set subsumed the other suggests that ideally both semantic and syntactic mutants should be used. Most of the semantic mutation operators are not more complex to apply in comparison to the traditional mutation operators.

The traditional operators just deal with the syntax, and the semantic mutants deal with semantics. So by using both of these operators we see that our tool SMT-P successfully kills both types of mutants.

5.2 Future Work

Following are the research directions in which work can be carried out.

- The proposed approach can be tested using larger test suites to enhance the quality of the software.
- Semantic mutation testing can be applied to a real life application.
- Additional semantic mutation operators can be developed.

Chapter 6 References

[1].Umar M., “An evaluation of Mutation Operators for Equivalent Mutants”, M.S. thesis, Computer Science, King’s College, london, 2006.

[2].Beizer B., “Software testing techniques”, Dreamtech Press, 2003.

[3].Pressman, Roger S., “Software Engineering: a practitioner’s approach”, Pressman and Associates, 2005.

[4]. D. D. Chamberlin and R. F. Boyce, “Sequel: A structured english query language,” in Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, ser. SIGFIDET ’74. New York, NY, USA: ACM, 1974, pp. 249–264. [Online]. Available: http://doi.acm.org/10.1145/800296.811515

[5].Google, “Google query language in Google app engine,” last access nov, 2013. [Online]. Available: https://developers.google.com/ appengine/docs/python/datastore/gqlreference

[6]. “Google query language in google cloud datastore,” last access nov, 2013. [Online]. Available: https://developers.google.com/datastore/ docs/concepts/gql

[7].Yahoo!, “Yahoo query language,” last access nov, 2013. [Online]. Available: http://developer.yahoo.com/yql/

[8]. EsperTech, “Event processing with wsper and nesper,” last access nov, 2013. [Online]. Available: “http://esper.codehaus.org/”

[9]. A. J. Offutt, A. lee, G. Rothermel, R. Untch, and C. Zapf. “An experimental determination of sufficient mutation operators”. ACM Transactions on Software Engineering Methodology, 5(2):pp. 99–118, April 1996.

[10]. J. Tuya, M. Suarez-Cabal, and C. de la Riva, “Mutating database queries,” Information and Software Technology, vol. 49, no. 4, 2007, pp. 398 – 417. [Online]. Available: “http://www.sciencedirect.com/ science/article/pii/S0950584906000814”

[11].D. Baldwin and F. G. Sayward, “Heuristics for determining equivalence of program mutations," tech report 276, Yale University, New Haven, Connecticut, 1979.

[12].Du Bousquet and M. Delaunay, “Towards mutation analysis for lustre programs". Electronic Notes in Theoretical Computer Science, vol. 203, no. 4, pp. 35-48, 2008

[13].N. Halbwachs, D. Pilaud, F. Ouabdesselam, and A.-C. Glory, “Specifying, programming and verifying real-time systems using a synchronous declarative language", in Proceedings of the international workshop on Automatic veri cation methods for nite state systems, (New York, NY, USA), pp. 213-231, Springer-Verlag New York, Inc., 1990.

[14].N. Halbwachs, F. lagnier, and C. Ratel, “Programming and verifying real-time systems by means of the synchronous data low language luster," IEEE Transactions on Software Engineering, vol. 18, pp. 785-793, Sep 1992.

[15].M. Ellims, D. Ince, and M. Petre, “The Csaw C mutation tool: initial results," in Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, (Washington, DC, USA), pp. 185-192, IEEE Computer Society, 2007.

[16].B. J. M. Grun, D. Schuler, and A. Zeller, “The impact of equivalent mutants," inProceedings of the IEEE International Conference on Software Testing, Veri cation, and Validation Workshops, (Denver, Colorado, USA), pp. 192-199, IEEE Computer Society, 2009.

[17].M. Harman, R. Hierons, and S. Danicic, “The relationship between program dependence and mutation analysis," in Mutation testing for the new century (W. E. Wong, ed.), Norwell, MA, USA: Kluwer Academic Publishers, pp.5-13, 2001.

[18].M. Woodward and K. Halewood, “From weak to strong, dead or alive? an analysis of some mutation testing issues", in Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on, pp. 152-158, Jul 1988.

[19].Y. Jia and M. Harman, “Higher order mutation testing," Information and Software Technology, vol. 51, pp. 1379-1393, Oct 2009.

[20].G. Kaminski and P. Ammann, “Using a fault hierarchy to improve the efficiency of DNF logic mutation testing," in Proc. Int. Conf. Software Testing Verification and Validation ICST '09, pp. 386-395, 2009.

[21].M. Kintis, M. Papadakis, and N. Malevris, “Evaluating mutation testing alternatives: A collateral experiment," in Proc. l7th Asia Pacific Software Engineering Conf. (APSEC), pp.300-309, 20l0.

[22].E. Martin and T. Xie, “A fault model and mutation testing of access control policies," in Proceedings of the l6th international conference on World Wide Web, WWW '07, (New York, New York, USA), pp. 667-676, ACM Press, 2007.

[23].K. Fisler, S. Krishnamurthi, l. A. Meyerovich, and M. C. Tschantz, “Verification and change-impact analysis of access-control policies," in Proceedings of the 27th international conference on Software engineering, ICSE '05, (New York, NY, USA), pp. l96- 205, ACM, 2005.

[24].J. Ofutt and Y.-R. Kwon, “The class-level mutants of MuJava," in Proceedings of the 2006 international workshop on Automation of software test - AST '06, AST '06, (New York, New York, USA), pp. 78-84, ACM Press, 2006.

[25].M. Papadakis and N. Malevris, “An empirical evaluation of the first and second order mutation testing strategies," in Proceedings of the 20l0 Third International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 'l0, , IEEE Computer Society pp. 90-99, 20l0.

[26].D. Schuler and A. Zeller, “Uncovering equivalent mutants," in Proceedings of the 3rd International Conference on Software Testing Verification and Validation (ICST'l0), (Paris, France), pp. 45-54, Apr 20l0.

[27].A. M. R. Vincenzi, E. Y. Nakagawa, J. C. Maldonado, M. E. Delamaro, and R. A. F. Romero, “Bayesian-learning based guidelines to determine equivalent mutants," Inter- national Journal of Software Engineering and Knowledge Engineering, vol. l2, no. 6, pp. 675-690, 2002.

[28]. Abdul Azim Abdul Ghani and Reza Meimandi,” Aspect-Oriented Program Testing: An Annotated Bibliography”, journal of software, vol. 8, no. 6, june 20l3.

[29]. Yue Jia and Mark Harman "An Analysis and Survey of the Development of Mutation Testing", IEEE Transactions On Software Engineering, vol. 7, no. 2, pp.77-84,2006.

[30]. J.A. Clark, H.Dan and R.M. Hierons ," Semantic mutation testing", Science of Computer Programming pp:345-363, 2013.

[31]. Haitao Dan and Robert M. Hierons "SMT-C: A Semantic Mutation Testing Tool for C" IEEE Fifth International Conference on Software Testing, Verification and Validation, 2012.

[32]. A.J. Offutt, "Investigations of the Software Testing Coupling Effect", ACM Transactions on Software Engineering and Methodology~, V(II 1. No 1, pp. 5-20, 1992.

[33].G. Fraser and A.Arcuri "Evo Suite: automatic test suite generation for object-oriented software" ESEC/FSE '11 Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 416-419,2011.

[34].M.Patrick, Manuel Oriol and John A. Clark "MESSI: Mutant Evaluation by Static Semantic Interpretation IEEE Fifth International Conference on Software Testing, Verification and Validation" 2012.

[35].S.Singh and S.Jain "A study on equivalent mutants detecting technique", VSRD International Journal of Computer Science & Information Technology, Vol. 3 No. 5 May 2013.

[36]. T.Y. Chen, T. H. Tse and Zhiquan Zhou, "Fault-Based Testing Without the Need of Oracles", Information And Software Technology, v. 45 n. 1, pp. 1-9,2003.

[37]. 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, 5(2):pp. 99-118, April 1996.

[38].E. S. Mresa and 1. Bottaci. "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. Software Testing, Verification and Reliability", 9(4):pp. 205-232, December 1999.

[39]. S. Hussain. Mutation Clustering. Master's thesis, King's College London, UK, 2008.

[40].J. R. Horgan and A. P. Mathur. “Weak Mutation is Probably Strong Mutation. Technical Report”, SERC-TR-83-P, Purdue University, West lafayette, Indiana, 1990.

[41].C. Ji, Z. Chen, B. Xu, and Z. Zhao. “A Novel Method of Mutation Clustering Based on Domain Analysis”. In Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09), Boston, Massachusetts, Knowledge Systems Institute Graduate School, 2009.

[42].A. J. Offutt and S. lee. “An Empirical Evaluation of Weak Mutation”. IEEE Transactions on Software Engineering, 20(5):pp. 337-344, May 1994.

[43].A. J. Offutt and S. D. lee. “How Strong is Weak Mutation?” In Proceedings of the 4th Symposium on Software Testing, Analysis, and Verification (TAV'91), Victoria, British Columbia, Canada. IEEE Computer Society, pp. 200 - 213,1991.

[44].V. Durelli, J. Offutt, and M. Delamaro. “Toward harnessing high-level language virtual machines for further speeding up weak mutation testing”. In Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, pp. 681 -690, April 2012.

[45].M. R. Woodward and K. Halewood. “From Weak to Strong, Dead or Alive? an Analysis of Some Mutation testing Issues”. In Proceedings of the 2nd Work- shop on Software Testing, Verification, and Analysis (TVA'88), Banff Albert, Canada, . IEEE Computer Society, pp. 152- 158, 1998.

[46].D. Jackson and M. R. Woodward. “Parallel firm mutation of Java programs”. In Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00), San Jose, California, 6-7, published in book form, as Mutation Testing for the New Century, pp. 55-61, 2001.

[47].P. R. Mateo, M. P. Usaola, and J. Offutt. “Mutation at system and functional levels”. In Proceedings of the 2010 Third International Conference on Software Testing, Verification, and Validation Workshops, ICSTW '10, IEEE Computer Society, pp. 110-119, 2010.

[48].K. N. King and A. J. Offutt. “A Fortran language System for Mutation-Based Software Testing. Software: Practice and Experience”, 21 (7):pp. 685-718, October 1991.

[49].M. E. Delamaro. Proteum – “A Mutation Analysis Based Testing Environment”. Master’s thesis, University of Sao Paulo, Sao Paulo, Brazil, 1993.

[50].M. E. Delamaro and J. C. Maldonado. Proteum – “A Tool for the Assessment of Test Adequacy for C Programs”. In Proceedings of the Conference on Performability in Computing Systems (PCS'96), New Brunswick, New Jersey, pp. 79-95, July 1996.

[51].B. Choi and A. P. Mathur. “High-performance Mutation Testing”. Journal of Systems and Software, 20(2):pp. 135-152, February 1993.

[52].R. A. DeMillo, E. W. Krauser, and A. P. Mathur. “Compiler-Integrated Program Mutation”. In Proceedings of the 5th Annual Computer Software and Applications Conference (COMPSAC'91), Tokyo, Japan, IEEE Computer Society Press, pp. 351-356, 2003.

[53].A. J. Offutt, Y.-S. Ma, and Y.-R. Kwon. “An Experimental Mutation System for Java”. ACM SIGSOFT Software Engineering Notes, 29(5):pp. 1-4, September 2004.

[54].Y.S. Ma, A. J. Offutt, and Y.-R. Known. “MuJava: An Automated Class Mutation System”. Software Testing, Verification & Reliability, 15(2):pp. 97-133, June 2005.

[55].D.Schuler, V. Dallmeier, and A. Zeller. “Efficient Mutation Testing by Checking Invariant Violations”. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'09), Chicago, Illinois, 19-23 July 2009.

List of publications

Ramil Gupta and Ajay Loura, "A Tool for Semantic Mutation Testing: SMT-P," Communicated at IEEE International Conference on Communication Control and Intelligent Systems (CCIS) - 2015, (2015)

Video Link

https://www.youtube.com/watch?v=CmphcSH4utU&feature=youtu.be