IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-6, Issue-7, July 2018 E-ISSN: 2347-2693

Mutation Operators in Python using SMT-P

R. Gupta[1*], C. Verma[2], N.Singh [3]

[1] Dept. of Computer Science and Engineering/BFCET, Bathinda, India
[2] Dept. of Computer Science and Engineering/BFCET, Bathinda, India
[3] Dept. of Computer Science and Engineering/BFCET, Bathinda, India

[*]*Corresponding Author: ramilgupta.bfcet@gmail.com, Tel.: +91-95011-15437*

Available online at: www.ijcseonline.org

Accepted: 03/Jul/2018, Published: 31/Jul/2018

Abstract— Mutation testing is one of the active techniques for testing its code. It is implemented by the replacement of the syntax of a program by another piece of code. This new version of the code is known as a mutant as is the most crucial part for testing the code. An effective test set is necessary to differentiate the mutant from the original program. In this paper we have presented a semantic mutation testing tool in python which works on semantics of python language than the traditional way of testing. In semantic mutation, a particular language is modified to create the mutant. Using SMT-P that is semantic mutation testing using python, we have inspected the efficiency of test cases on both traditional and semantic mutation operators. Comparison of traditional and semantic mutation testing operators in same test cases has been found to be quite useful and proves the usefulness of Semantic based testing over traditional one.

Keywords— Mutation testing, SMT (Semantic mutation testing), SMT-P (Semantic mutation testing tool in python), Language

I. INTRODUCTION

Testing plays the most essential role of the several levels of software development life cycle. It is a method of determining faults. The testing is very effective and the quality of software is higher. The correct execution of a program is ensuring by testing in each probable scenario by creation of active test cases which identify the possible faults.

Mutation testing is a technique aimed at locating and exposing the weakness in test suites [9-10]. To evaluate the effectiveness of test set is the main focus. here the program structure is considered for detection of faults like in white box testing.

Mutation testing is also mentioned as fault based testing. Any small change which distinguishes the program from the original program is a mutant. There are several types of mutants: stillborn, trivial, equivalent.

II. RELATED WORK

The prerequisite is a source code and a test suite for that source code. To produce a mutant, one and only thing which is required is to vary the original program by inserting a minor fault in it. By running the original test data this modification is tracked. The change in the original code has been detected (dead/ Killed Mutant) is states by differences between the original and the changed code. In case the mutant remains alive, the possibility arises either if the mutant and the original program are identical or the mutant could not be killed as the test set was insufficient which was incapable to identify the made conversion.

For mutation traditional mutation testing consists of operators that represent syntactically small faults like exchanging + by – in an arithmetic expression.

Traditional mutation testing are several weaknesses. There are listed Some of the drawbacks: for a small program, the number of mutants produced is large [8], with this the probabilities of equivalent mutants are rises. Additional amount of manual work is necessary, in order to deal with equivalent mutants. The cost of testing is rises by this additional determination. The confusions related to semantic changes do not consider by Mutation operators, syntactic level is the only concern. To resolve the above problems automatic detection of equivalent mutants and lowering the introduction of equivalent mutants has been a major concern [5-8]. In this paper presents Semantic Mutation testing and a tool (SMT-P) designed in python for semantic mutation testing.

III. METHODOLOGY

In order to deal with several specific types of mistakes, Semantic Mutation Testing was proposed [2], [4]. A small change in syntax can have a large effect on semantics [3]. For introducing the semantic mistakes, different ways are available [4]. For SMT-P, modification in the syntax of description has been selected in order to simulate semantic mutation.

There are three consequences of mutation testing, strong MT, weak MT and firm MT. The program and the mutant can be individually recognized in the case of solid mutation testing, if they produce dissimilar outputs for a similar test case. On the other hand, in weak mutation testing, the program and the mutant are illustrious when they produce a dissimilar value for a specific variable, at a specific point in the program [1]. When the tester himself picks the point where the dissimilarity in the value of a variable should reflect is firm mutation testing [5].
We study that in a language L and description N the programs to be written. The behaviour of a program is defined by the mixture of description and language i.e; (M, L). In traditional mutation testing, there is alteration in the syntax, the mutant of the above description can be (M', L). On the other hand, the description can be denoted for semantic mutation testing as (M, L').

We have developed a new mutation testing tool for Python, called SMT-P. The development of SMT-P inspired from the fact, that there is no such tool that fulfil the necessities of SMT. The simulations in the semantic mutations return with the alterations made in the syntax of the description. The aim was to design an easy to use and a flexible tool. The semantic mutation operators have been defined in this specific unit.

ARCHITECTURE OF SMT-P AND EXPERIMENTAL STUDY

SMT-P is a tool developed in Python. This can run independently or using Pycharm[11].

```
...
Select the source file of which you want to genrate mutant
You have selected the C:/Users/bond/Desktop/R thesis/6.version/TCAS.py
Which mutant you want to apply
Enter 1 for if else mutant
Enter 2 for break to continue
Enter 3 for continue to break
Enter 4 for provide default for switch statement
Enter 5 for  modify last case of switch
Enter 6 for  floor of division
Enter 7 for  ceil of division
Enter 8 for Indentation mutant
Enter 9 for the el-If mutator
```

Figure 1: Choosing the input program

To choose any option here a choice menu is provided to the user. There are nine choices shows in menu. A mutant is created accordingly from which one is selected. To see whether the mutant gets destroyed or not the test cases can be applied.

Test viewer : This is a front end to see the test suites associated to a specific program. The outcomes can be observed by clicking on a specific test case.
This gives expected result, a complete view of results, the exact errors given by unit test cases eg. assertion errors etc., picturing the exact parameters passed and result obtained.

The function components have been designed in order to build, test and execute mutants.

THE MUTATION GENERATOR

1. According to the ideas in [11], there are seven semantic mutation operators have been implemented. If – Else: To those 'if' constructs which do not have an else branch, an else branch is added.
2. Last case of switch: a default branch is added, when using a switch case without a default branch.
3. Default for switch: Here the previous branch of switch statement is changed to be the default statement of switch.

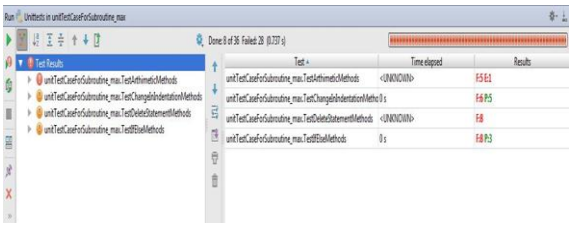

Figure 2: Test Viewer

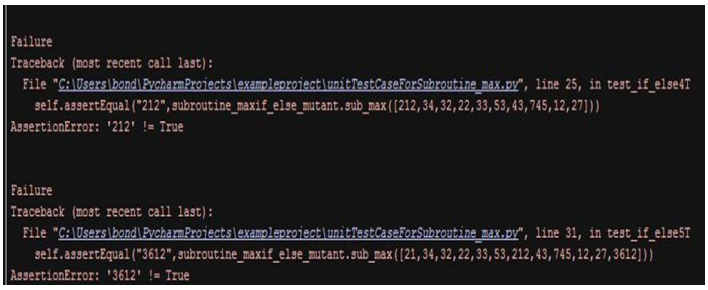

Figure 3: Console viewer

To confirm complete branching structures, the above two operators are designed. As the programmers could assume the implementation of last branch o structure, because Unfinished branches can lead to faults.

4. Floor of division: Using the floor method, Truncating the float value.
5. Ceil of division: The quotient depends on the approach defined, on division operations i.e. the result can either be the just the immediate earlier integer of the quotient or the immediate next integer to the quotient. Here to form the mutant the tail method is used.
6. Indentation: This plays a main role in python. To see the result of test cases, Here the indentation of

statements is changed. As there are no starting and closing brackets.

7. Elif: Wherever in a 'if' statement followed by an 'else- if' statement, else is missing, an else statement is added.

IV. RESULTS AND DISCUSSION

From table 1 and 2, we see that to produce mutant , each program we have applied seven semantic mutation operators on some programs. To check the effectiveness of the test suites, the specific test suites have been applied to the mutant and the program. Ex: In Tcas, entire of 10 test cases have been applied out of which all the test cases are capable to distinguish the mutant from original program for the mutant produced by 'if - else' operator. As outcome, we get 100 % efficiency of the test suite. Equally, the efficiency has been checked upon for other programs also. For Trim test suite is 60% efficient, or for tcas test suite is 100% efficient. (killed/total) *100 is calculated to calculate the efficiency. Similarly, also the efficiency has been calculated for traditional mutation operators. By matching the results, SMT-P is more efficient for semantic mutation testing.

Table 1: Efficiency Of Test Suites In Smt

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 2: Efficiency Of 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	

V. CONCLUSION AND FUTURE SCOPE

In this paper, we have presented a new tool SMT-P based on semantic mutation testing. Regarding of the semantics of the description language there can be various misunderstandings. In traditional mutation testing the syntax of a description is mutated. In other side, when we deal with language, it is semantic mutation testing. when a test case run on actual program, that produces different results and on its mutant is said to be failed. The mutant is said to be killed, When a test case fails. For different operators, here we have calculated the efficiency of different test suites. As a part of future work, the alike can be tested for larger test suites. Extra semantic mutation operators can be established.

REFERENCES

[1] W.E. Howden, Weak mutation testing and completeness of test sets, IEEE Transactions on Software Engineering 8 (4) (1982) 371–379

[2] A.J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering Methodology 1 (1) (1992) 3–18.

[3] R.G. Hamlet, Testing programs with the aid of a compiler, IEEE Transactions on Software Engineering 3 (1977) 279–290.

[4] K.S.H.T. Wah, a theoretical study of fault coupling, Journal of Software Testing, Verification and Reliability 10 (1) (2000) 3–45.

[5] A.J. Offutt, J. Pan, Detecting equivalent mutants and the feasible path problem, in: Annual Conference on Computer Assurance, COMPASS 1996, IEEE Computer Society Press, Gaithersburg, MD, 1996, pp. 224–236.

[6] R.M. Hierons, M. Harman, S. Danicic, using program slicing to assist in the detection of equivalent mutants, Journal of Software Testing, Verification and Reliability 9 (4) (1999) 233–262.

[7] J.A. Clark, H. Dan, R.M. Hierons, Semantic mutation testing, in: Fourth Workshop on Mutation Analysis, 2010, pp. 100–109.

[8] J. Offutt, J. Pan, automatically detecting equivalent mutants and infeasible paths, Software Testing, Verification, and Reliability 7 (3) (1997) 165–192.

[9] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practical programmer, IEEE Computer 11 (4) (1978) 31–41

[10] A.J. Offutt, J.H. Hayes, A semantic model of program faults, in: International Symposium on Software Testing and Analysis, ISSTA 1996 1996, pp. 195–200.

[11] "Pycharm python IDE" https://www. jetbrains. com /pycharm/.

[12] L. Hatton, Safer C: Developing Software in High-integrity and Safety-critical Systems, McGraw–Hill, 1994.

