Volume 6, Issue 2, April 2019 ISSN: 2348-8565 (Online)
International Journal of Modern Engineering and Research Technology
Website: http://www.ijmert.org Email: editor.ijmert@gmail.com

Bebugging: An Approach for Mutation Testing

Venkateshwara Reddy Mudiyala
Research Scholar
Master of Computer Science
University of New Haven.
West Haven, (CT) [UNITED STATE]
Email: Vmudi1@unh.newhaven.edu

Henry J Nowik
Professor
Department of ECECS,
University of New Haven.
West Haven, (CT) [UNITED STATE]
Email: *HNowik@newhaven.edu*

ABSTRACT

Be Bugging/Mutation Testing is a way of determining the effectiveness of testing. That is, it is a technique that may be used to determine the number of remaining bugs in a software artifact after testing/review. Mutation Testing is a fault-based testing technique which provides a testing criterion called the "mutation adequacy score". The mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults

Keywords :— *Mutation testing, Decision Mutation, Statement Mutation, Operators*

I. INTRODUCTION

Mutation testing, proposed in 1978 by Richard A. DeMillo and his colleagues2 is an effective technique: if a test suite finds all the artificial errors inserted in the mutants and finds no fault in the original, it's likely that the program under test is free of them. Obviously, the validity of this affirmation depends on the nature of the artificial fault: some of them are better than others. This testing technique has been used in the research arena to check the effectiveness of new proposed testing techniques, but it hasn't been used until recently in industry due to its costs and the lack of knowledge and industrial tools.

Mutation Testing has been increasingly and widely studied since it was first proposed in the 1970s.There has been much research work on the various kinds of techniques seeking to turn Mutation Testing into a practical testing approach. It is a type of White Box Testing which is mainly used for Unit Testing. The changes in mutant program are kept extremely small, so it does not affect the overall objective of the program. The goal of Mutation Testing is to assess the quality of the test cases which should be robust enough to fail mutant code. This method is also called as Fault based testing strategy as it involves creating a fault in the program.

II. MUTATION TESTING TYPES

Mutation testing can be broadly classified into three – Value mutation, decision mutation and statement mutation. In value mutation, value of constants or parameters is changed. For example, value is changed to one larger or one smaller in loops, Initialization value is changed.

Decision mutation – This helps to modify program code so that slip errors are reflected. For example, > a is changed to <a.

Statement mutation – In this type of testing, developer cuts and paste codes which might result in deletion of some

statements of lines. This may also involve swapping the order of line of code. A line of code can be deleted/duplicated. Order of statements can also be changed

III. Mutation Operators

The operators that are applied on the original program to generate the mutants are known as Mutation operators. The Mutation operators can be broadly classified into

1. Traditional mutation operators
2. Class mutation operators

1. Traditional Mutation Operators

The traditional mutation operators are developed for procedural programming language. Though application of these operators generates many mutants, all of them may not be effective as they tend to overlap. The following are the traditional mutation operators.

i. Arithmetic Operators
ii. Relational Operators
iii. Conditional Operators
iv. Logical Operators
v. Assignment Operators
vi. Shift Operators

The mutants are generated by replacing, inserting or deleting the mutant operators.

2. Class Mutation Operators

These are used for generating mutants to test object -oriented and integration issues.

(i) Encapsulation: Mutants are formed by application of operators that modifies, deletes or insert the access level for instance variables and methods.

(ii) Inheritance: The mutants are produced by application of operators that deletes a hiding variable to check whether that variable is defined and that its accessibility in class and subclasses are correct.

(iii) Polymorphism: The mutants are created by the polymorphism operators to check if the methods having the same name and number of parameters are accessible in a right manner or not.

(iv) Mutation Operators/mutators: A mutator is the operation applied to the original code. Basic examples include changing a '>' operator by an '<', replacing 'and' by 'or' operators, and substituting other mathematical operators for instance.

(v) Mutants: A mutant is the result of applying the mutator to an entity (in Java this is typically a class). A mutant is thus the modified version of the class, that will be used during the execution of the test suite.

(vi) Mutations killed/survived: When executing the test suite against mutated code, there are 2 possible outcomes for each mutant: the mutant is either killed, or it has survived. A killed mutant means that there was at least 1 test that failed as the result of the mutation. A survived mutant means that our test suite didn't catch the mutation and should thus be improved.

(vii) Equivalent Mutations: Things are not always white or black. Zebras do exist! On the mutation testing subject, not all mutations are interesting, because some will result in the exact same behavior. Those are called equivalent mutations. Equivalent mutations often reveal redundant code that may be deleted/simplified.

IV. Execution of Mutation Testing

How the mutation testing will take place explained below step by step

1. Faults are introduced into the source code of the program by creating many versions called mutants. Each mutant should contain a single fault, and the goal is to cause the mutant version to fail which demonstrates the effectiveness of the test cases.
2. Test cases are applied to the original program and to the mutant program. A Test Case should be adequate, and it is tweaked to detect faults in a program.
3. Compare the results of an original and mutant program.
4. If the original program and mutant programs generate the different output, then that the mutant is killed by the test case. Hence the test case is good enough to detect the change between the original and the mutant program.
5. If the original program and mutant program generate the same output, Mutant is kept alive. In such cases, more effective test cases need to be created that kill all mutants.

There are several Mutation testing tools that are available. Jumble and Insure++ are the most common among that.

V. Mutation Testing Tools

5.1 Jumble

1. Jumble is a simple non-graphic open source tool. It converts the text files into version that enables studying the format of the file.
2. It directly operates at a source code level and speed up the Mutation testing process. The limited sets of Mutation operators supported by Jumble are: Conditional, Binary Arithmetic Operations, Increments, Inline Constants, Class Pool Constants, Return Values, and Switch Statements

5.2 Insure++

1. It is a commercial automatic testing tool for C and C++ that makes use of Mutation analysis technique.
2. Instead of generating all possible mutants, insure++ focuses on the "potential equivalent mutants". The motivation behind this idea is that if any test case can kill the "potential equivalent mutants", it might also find the faults in the original program.

VI. Challenges of Mutation Testing

Mutation testing can effectively assess the adequacy and quality of a test set, but it also has certain challenges as below

1. Mutation testing has a high computational cost of executing the enormous number of mutants against a test set.
2. The human oracle problem, which refers to the process of checking the output of each test case against the output of original program, can be a serious problem as mutation testing can lead to an increase in the number of test case

VII. Advantages

- It is a powerful approach to attain high coverage of the source program.
- This testing is capable comprehensively testing the mutant program.
- Mutation testing brings a good level of error detection to the software developer.

- This method uncovers ambiguities in the source code and has the capacity to detect all the faults in the program.
- Customers are benefited from this testing by getting a most reliable and stable system.

VIII. DISADVANTAGES

- Extremely costly and time-consuming since there are many mutant programs that need to be generated.
- Since its time consuming, it's fair to say that this testing cannot be done without an automation tool.
- Each mutation will have the same number of test cases than that of the original program.
- So, many mutant programs may need to be tested against the original test suite.
- As this method involves source code changes, it is not at all applicable for Black Box Testing.

IX. CONCLUSION

It is the most comprehensive technique to test a program. This is the method which checks for the effectiveness and accuracy of a testing program to detect the faults or errors in the system.

REFERENCES:

[1] Mutation Testing for the New Century Editors: Wong, W. Eric (Ed.)

[2] R. T. Alexander, J. M. Bieman, S. Ghosh, and B. Ji, "Mutation of Java Objects," in Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02). Annapolis, Maryland: IEEE Computer Society, 12-15 November 2002, pp. 341–351.

[3] 3.B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. Le Traon, "Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment," in Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02), Annapolis, Maryland, 12 - 15 November 2002, pp. 195–206.

[4] B. Bogacki and B. Walter, "Evaluation of Test Code Quality with Aspect-Oriented Mutations," in Proceedings of the 7th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP'06), ser. LNCS, vol. 4044, 2006, Oulu, 17-22 June 2006, pp. 202–204.

[5] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation Operators for Concurrent Java (J2SE 5.0)," in Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION'06). Raleigh, North Carolina: IEEE Computer Society, November 2006, pp. 83–92.

[6] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer," Computer, vol. 11, no. 4, 1978, pp. 34–4

[7] M. Polo, M. Piattini, and I. García-Rodríguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," Software Testing Verification Reliability, vol. 19, no. 2, 2009, pp. 111–131.

[8] B. K. Aichernig, "Mutation Testing in the Refinement Calculus," Formal Aspects of Computing, vol. 15, no. 2-3, pp. 280–295, November 2003.