AN ANALYSIS OF MUTATION TESTING

Ruchi Goel, Kriti Seth, Richa Sharma
Department of Computer Science,
Maharaja Agrasen Institute of Technology, GGSIPU, New Delhi.
kritiseth20@gmail.com, richa101994@gmail.com

Abstract-Mutation testing is regarded as an important technique used to iteratively refine the test data and hence build up confidence in the system. It has a lot of advantages as far as fault revealing is concerned. Despite of this fact, it is not used much in the industry of software development because of the expenses included in it. Discovery of the most efficient and optimized automated tools for mutation testing is thus, the need of the hour. There is a lot of evidence to prove that techniques that automatically generate test data can act as an important catalyst in the automated testing process. However, very little work has been done in applying them in the context of mutation testing. In this paper, results from two tools applied for mutation testing are analyzed and discussed. It also discusses some cost reduction techniques that can be applied in the framework of this testing technique. Additionally, a system is proposed to improve the generation of effective test cases.

Keywords — Mutation, Test Cases, mutation score, equivalent mutants, Junit, MuClipse, Jumble, Clustering, sampling, genetic algorithm

1. INTRODUCTION

One of the most important phases of the software development cycle is the testing phase which requires a lot of time, effort and expense. Consequently, it is vastly required that the testing activity incorporates effective and efficient methods bagged up with the utmost level of automation available. Test data generation plays a very crucial role in achieving this efficacy in the testing procedure. The current scenario is such that it doesn’t incorporate the level of automation as it is ought to be. Hence, it is required to produce the test data with efficient automation so as to intensify the thoroughness of the system under test apart from reducing the cost involved. Testing quality is usually measured by the test adequacy criteria. Some tests applied to the data under consideration are better than others, but it is often difficult to identify which of these are. There is no means to measure the test’s effectiveness. The key is to know whether faults exist prior to testing, but knowing this makes testing redundant. To aid with this paradox, criteria are used to provide a requirement for test data adequacy, and so give a measure for improving a test set. If the test data is in compliance with a criterion then it stands adequate for testing purpose, and hence more likely to indicate faults if they exist. However, typical criteria do not focus on the cause of a program’s failures - the faults, the mutation adequacy criterion does. A test set will not be able to identify all faults, in which case the mutation adequacy criterion gives a measure to determine the effectiveness of a test case and basis for selecting a new test set.

2. MUTATION TESTING

Mutation testing or mutation analysis, is a fault-based technique introduced by Hamlet and DeMillo et al. Mutation analysis involves making modifications to the code under test based on a set of simple syntactic rules called mutant operators. Mutation testing is developed based on a very simple rule of “who will guard the guards”. It is a process of deliberately introducing faults into the program and each altered version of the original code is called a mutant. Examples of mutants are: negating a condition in an If statement, changing a conditional boundary in a For loop, or throwing an exception at the end of a method.
quality. Basically, the aim is that the test case which is put into action is able to reveal the mutants, which are termed as “killed” if they are detected. This mutant is of no value and thus is killed, but the test case is important as it can distinguish incorrect programs and is kept. The higher the percentage of the mutants that get killed by the unit tests, the better will be the quality of the test suite. When the system is not able to identify this variation in the form of a mutant, it is said to have “lived”. Once all test cases have been executed on all mutants, those mutants that remain alive are indistinguishable from the original because they produce the same output. These are known as equivalent mutants and are removed from the mutation testing process before proceeding. Equivalent mutants are analogous form of the infeasibility element problem faced in structural testing. The Testing capability in the course of mutation testing is measured using the mutation score which is defined as the ratio of the number of killed mutants to the entire number of applicant mutants reduced by the number of equivalent ones.

The strength of this method relies on the hypothesis - ability of the introduced mutants to produce realistic faults. Apart from this it is also based two other hypothesis. The Competent Programmer Hypothesis states that the programmers are proficient enough to create correct versions of the programs. They do not create arbitrary programs. This hypothesis assumes that only simple faults are done by programmers which can be removed by making elementary syntactic changes to a program. The coupling effect implies that tests that can distinguish small errors are implicitly able to recognize complex errors as well, because they are actually very sensitive to alterations. Complex faults and simple faults are coupled and are detected in mutation testing.

Mutation testing has been widely studied since it was first proposed in the 1970s. There has been a lot of research work on the various kinds of techniques seeking to turn it into a practical testing approach. It is evident that developers can benefit from applying mutation testing because it is a powerful approach. Unfortunately, there are few major troubles that prevent engineers from realizing its true value: High computational cost of executing the enormous number of mutants and the human oracle problem of checking the original program’s output with each test case. In addition to this, finding the mutations is also expensive. These constitute the main issues of the present research.

![Figure: The process of Mutation Testing](image-url)
3. AUTOMATION TESTING OVER MANUAL TESTING

Taking tool support and executing the test cases by using automation tool is considered to be a more sustainable option than executing the test cases manually for varied reasons:

1) **Time**-Automated techniques run test cases significantly faster than human resources who rather make it slow and tedious.
2) **Investment cost**-Since the test cases need to be executed physically in the manual process, more testers are required in it adding to the cost.
3) **Reliability**-Manual testing is less trustworthy than its automated counterpart because tests may not be executed with precision each time because of human errors.
4) **Programmability**-In the case of automation testing, the testers can write down sophisticated tests to bring out some hidden information which is not achievable when done manually.

4. MUTATION TESTING- AN ANALYSIS

4.1. Test cases and test suites using JUnit

The analysis has been done for a sample java program (To find out the type of triangle among scalene, isosceles and equivalent) and its corresponding JUnit test cases. JUnit being an instance of the xUnit architecture for unit testing framework is vital in the test driven development and provides a simple unit testing structure to write repeatable tests. JUnit upholds the idea of "first testing then coding", which basically requires setting up the test data for a piece of code first, which is then tested and later implemented.

In the analysis, JUnit was embedded in the Eclipse Ganymede (a popular IDE or integrated development environment), and was used to create test cases as well as test suites. JUnit is linked with the IDE as a JAR at compile-time.

JUnit has been used because it is the de facto unit testing framework for the Java language. And working with a code in java makes performing mutations at either the source code or the byte code feasible. A class and its corresponding JUnit test is a workable granularity at which to apply mutation testing.

4.2. Creating and running mutants using MuClipse

MuClipse is an Eclipse Plugin which provides developers with a bridge between the existing MuJava mutation engine and the Eclipse IDE and is used to create mutants that are variations of the original program. The configurations that come up with the muclipse installation handle a lot of specific java runtime settings. This is required because the mutation process requires a large amount of meta-language manipulation. Mutation operators act on the source code and produce mutants which are then required to be compiled before it can be run against the test cases. Also, mutation testing requires that the mutant's source code not be on the Java classpath when attempting to "kill" it. MuClipse involves two types of mutation operators:

1) **Traditional**-Works only at the method level and to inject faults at this level MuClipse inserts, replaces or deletes primitive operators in the
program (For example: switching operands, replacing + with -).
2) **Class-Level**-It works at the class level and involves altering keywords that give the type of class or the methods involved (For example: overloading a given method, changing a class to static).

Muclipse v1.3 version was used which is compatible with the version of eclipse used that is eclipse v3.4 (Ganymede). It provides two run time configurations. The first one uses a given set of mutation operators (traditional or class level operators) on the given java source code to create mutants, which it stores in a folder within the same project. The second program runs a given test or set of tests on the original code and the generated mutants and compares the results. After all the results have been tabulated, MuClipse returns a mutation score which is the percentage of all the generated mutants that have been killed.

4.3. Using Jumble for checking efficiency and coverage of Junit test suites

Jumble is another java mutation tool which was used in the analysis for measuring the efficacy and coverage of JUnit test suites. It basically gives an answer to the question as to how good the JUnit test cases are. Jumble is a class level mutation testing tool that works in unification with JUnit. Jumble gives the worthiness of the code from 0% (worthless) to 100% (angelic).

Jumble is a new mutation tester operating directly on class files, at the byte code level and runs the corresponding unit test in order to find out the number of mutants that have been killed. It uses the byte-code engineering library (BCEL) to directly modify class files thereby drastically cutting the time taken for each mutation test cycle. After all the results have been tabulated, Jumble returns a mutation score which is a percentage of all the generated mutants that have been killed. Jumble is required to be loaded into Eclipse IDE and it is also required to be pointed to the project on which it will perform mutation testing.

4.4. Comparing MuClipse with Jumble

Both Jumble and MuClipse work for the same purpose of mutation testing that is to provide a measure of the effectiveness of the test cases generated by JUnit. A single mutation is performed on the code to be tested, the corresponding test cases are then executed. If the modified code fails the tests, then this increases confidence in the tests. Conversely, if the modified code passes the tests this indicates a testing deficiency. Jumble and Muclipse were used to generate mutants and check their efficiency. Jumble is a new mutation tester which works directly on class files. It uses the byte-code engineering library (BCEL) to directly modify class files and therefore radically cutting the time taken for each mutation test cycle. MuClipse on the other hand uses Junit to create test cases and then creates mutants explicitly unlike jumble which creates mutants implicitly from the code. This is the major reason behind the fact that Junit reduces the time required for each mutation cycle but it gives a less mutation score of 66% for the program under test (To find out the type of triangle among scalene, isosceles and equivalent). On the other hand, Mucclipse takes more time than jumble but it gives a better mutation score for the same program and junit tests. It gives a score of 72%.
5. REDUCING COMPUTATIONAL EXPENSE

In order to turn Mutation Testing into a practical testing technique, many cost reduction techniques have been proposed. Cost reduction techniques are divided into three types: ‘do fewer’, ‘do faster’, and ‘do smarter’. Cost can be reduced by either reducing the generated mutants or by reducing the generation of test cases. In our paper we basically focus on the mutant reduction techniques.

5.1. Mutant Sampling

This is an idea proposed by Acree and Budd and is used to reduce the range of mutants. It is a very simple approach that randomly chooses a small subset of mutants from the entire set. All the possible mutants are generated using the traditional mutation testing, then x% of these mutants are selected arbitrarily for mutation analysis and the remaining mutants are discarded. Some studies suggest that random selection of 10% of mutants is only 16% less effective than a full set of mutants in terms of mutation score and hence, this technique doesn’t render the test much less efficient.

5.2. Mutant Clustering

The concept of mutant clustering was first proposed in Hussain’s masters thesis. In this method instead of selecting mutants randomly as in the case of mutant sampling, subsets of mutants are selected using clustering algorithm like k-mean and agglomerative clustering. A set of all the possible mutants are generated initially, and then the clustering algorithm is applied to classify the first order mutants into diverse clusters based on the killable test cases. Only a small number of mutants are then selected from each cluster for mutation testing and the remaining are discarded.

5.3. Selective Mutation

A reduction in the number of mutants can also be accomplished by reducing the number of mutant operators applied; this was first suggested by Mathur and was implemented as “2-selective mutation”. We can find a small set of mutant operators that generate a subset of all possible mutant operators that too without any significant loss in the test efficiency. In order to reduce the number of generated mutants effectively, Mathur suggested the removal of two mutation operators: ASR (Array reference for scalar variable replacement) and SVR (scalar variable replacement) which generated most of the mutants.

5.4. Higher Order Mutation

When the mutation operators are applied only once on the source code we get first order mutants (FOM). While, when we apply the operators more than once we generate high order mutants (HOM). To achieve effective testing strategy, the main aim is to find rare but valuable higher order mutants that represent subtle faults. It is preferable to replace FOMs with the single HOM to reduce the number of mutants. Also, a subsuming HOM is harder to kill than the FOMs from which it is constructed.

6. FURTHER STUDIES

Proposed Model for Mutation Testing Using Genetic Algorithm

The proposed framework applying genetic algorithm to the traditional mutation testing process starts by first instrumenting the
original source program and the mutant programs so that each program can be viewed as an individual small unit. Rather than checking the entire program at one time, the proposed system executes these small units individually and attempts to kill each mutant unit. If a data state at the mutant object and original program are not equal then it starts testing again with new test data generated by genetic algorithms.

6.1. Genetic Algorithm

Genetic Algorithm is a search algorithm based on the system of natural selection and natural genetics. In the context of software testing or more precisely mutation testing, the basic idea to incorporate it, is to search the domain for input variables, which satisfy the goal decided for testing. The genetic algorithm searches the input domain of the original program for suitable test cases to kill a given mutant. A genetic algorithm uses three basic operators: Selection, Reproduction and Mutation.

**Selection:** It is the process that involves choosing two individuals from a generation for the recombination process (crossover and mutation). Individuals can be selected in two ways: randomly or with respect to their fitness value. If the latter is applied, then the individuals with a higher fitness value have a higher probability of being selected whereas the others are more likely to be discarded.

**Reproduction:** It is a process following the selection process. Reproduction involves processing the individuals according to their fitness value using crossover and mutation operators. This step is the key to the power of genetic algorithms. The operators create new individuals with the idea that the new individuals will be closer to a global optimum. Applying the Crossover operator first requires the individuals to be converted into a binary representation. During a crossover, the two parents exchange substring information at a random position in the individual to produce two new strings (offspring). The main motive here is to create better individuals and a better population over time by combining material from pairs.

**Mutation:** The mutation operator modifies one or several genes’ value. The reproduction and crossover operators are so powerful in improving the search that the mutation operator usually plays a secondary role, i.e., it modifies the value of the test methods.

6.2. Genetic algorithms with a checker

Sometime a mutant survives even when there is a state difference between the original and the mutant program at the mutation statement because this difference is not transferred to the output. We use a strategy where the source program and the mutant programs are developed in such a manner that the input as well as output behavior of each unit can be traced. We then trace the execution of a test case and record data states of the original as well as the mutant program. A checker module is used to compare and trace the output of each unit. The checker then writes 1 in a log if the mutant unit lives and writes 0 if the mutant unit is dead. In this way we continue to develop the final output pattern of the entire program. We can then locate the faulty unit by looking at the position where 1 appears.

Hence, using the genetic algorithms and a checker, we can calculate the better test cases which improve the difficulty of having many test cases to run against mutants. Thus, genetic algorithms improve the generation of effective test cases and hence reduce the cost and time of executing the
test cases over mutated programs.

Figure: Genetic Algorithm Supported With A Checker

7. REFERENCES


