Software Testing and Reverse Engineering
Mutation Testing
Papers Summary

February 29, 2016

Raies Saboerali - 4080335 - r.a.a.saboerali@student.tudelft.nl
Samuel Austin - 4005996 - s.e.austin@student.tudelft.nl
Brynjolfur Mar Georgsson - 4404548 - b.m.georgsson@student.tudelft.nl

Contents

1 Introduction

Mutation testing was first proposed by DeMillo and Hamlet [6]. The idea is to alter a program and test it. By altering the program, the meaning of the program should be broken and one or more tests results should change. If this does not happen, then either the tests are not good enough or there is a fault in the program. Mutation testing can be used in order to improve the tests or identify problems in the program which would have been much harder to detect with traditional testing.
The number of mutations killed in each run is expressed as a ratio of the total number of mutants excluding equivalent mutants. The goal is to achieve a mutation score of 1. If the test result changes, the mutant is said to be killed. If not, the mutant is said to have survived, these are the mutants to look at. However, some mutants may not be killed because of mathematical equality, such mutants are called equivalent mutants.

2 Mutation Operators

Mutation operators are the set of rules which are used to make a few syntactical changes to parts of the code. This can include, changing return values, removing statements, changing Boolean operator etc. If one has a program p, then p' is called the faulty program which is generated after applying a mutation operation. Jia and Harman present 22 Mothra FORTRAN operators [6]. These operators were specifically for FORTRAN. For various programming languages their are slightly different operators. Some operators might not work in all languages because of the design and restrictions of a language.
In mutation testing, mutations can be classified into First Order and Higher Order Mutants. First Order Mutants are created by applying the mutation operator once, whereas Higher Order Mutations are created by applying more than one mutation operation.
An example:

```
if (a > 2 && b < 3) {
    return a;
} else {
    return b;
}
```

After applying a mutant operator once:

```
if (a > 2 || b < 3) {
    return a;
} else {
    return b;
}
```

2.1 High Order Mutants

Jia and Harman introduce the concept of subsuming HOMs, which are harder to kill than the FOMs it is constructed from [5]. Assume a subsuming HOM h and constructed from the FOMs $f_1, ..., f_n$. The test cases that kill h, also kills each FOM. However, the converse does not hold: there exists a test set that kill all the FOMs, but not the HOM h.
A HOM is said to be strongly subsuming if it is killed by a subset of intersection of test cases that kill each FOM of which it is constructed [5]. If this is not the case, it is called a weakly subsuming HOM.
A HOM is said to be coupled if the test set that kills the FOMs also contains cases that kills the HOM. If this does not hold, then the HOM is said to be decoupled.
From this six possibilities can be considered: Strongly subsuming and coupled, weakly subsuming and coupled, weakly subsuming and decoupled, non-subsuming and decoupled, non-subsuming decoupled (equivalent), and non-subsuming and coupled.
If the system has n FOMs then the possible HOMs are n^n. This is a lot, however this number can be reduced by looking at the HOMs, especially subsuming and strongly subsuming HOMs which are more useful.
As stated, strongly subsuming HOMs (SSHOMs) consists of a subset of FOMs that are killed. This means that using SSHOMs reduce the testing effort without loss of effectiveness [6]. The study also suggests that around 15% of all subsuming HOMs may be strongly subsuming.
It is believed that HOMs offer the following benefits:

- Increased subtlety: the majority of FOMs are killed by simple faults, such are removing frequently used statements.
- Reduced test effort: there are more HOMs than FOMs. But strongly subsuming HOMs can replace multiple FOMs, which means fewer mutants and fewer test cases. However, one needs to select subtle HOMs in order to be most effective.
- Reducing number of equivalent mutants: this is an undecidable problem, however there are approaches in order to detect equivalent mutants, e.g. a search based algorithm.

A fitness of a HOM/FOM is defined as how easy a HOM/FOM is killed. In order to compute fitness, fragility is defined. Fragility of a mutant is computed by the total number of test cases that kill the mutant over the total amount of test cases. The value can be between 0 and 1. 1 means that the mutant is so weak, it can be killed by "anything". 0 means that there is no test case that kills the mutant, this indicates that the mutant might be equivalent. It may also mean that the mutated part is not tested. It is up to the tester who needs to manually check this.
The fitness of a HOM is defined as a ratio of the fragility of its HOM to the fragility of the FOMs it consists of. If the fitness is greater than 1, than the HOM is weaker than the FOMs. But as the value decreases from 1 to 0, the HOM becomes stronger than its constituent FOMs. A '0', may be an indicator of an equivalent HOM.

2.2 Inter-class Mutation Operators

A lot of mutation operators have been developed for conventional programs, however these operators are not that well designed for use with Object-oriented programming [9]. OO-programming introduces new features as encapsulation, inheritance, and polymorphism. These new features introduce new types of errors to be made while programming. A few errors that can be made are: "this" or "super" keyword misuse, access modifier misuse, overloading method misuse etc. In [9] a set of operators can be found that deal with these kind of faults that can be made in OO-programming.

3 Cost reduction

Each mutation that is applied to the source code requires the source code to be compiled and the unit tests to be run in order to evaluate that mutation. The high number of mutations that can be applied to the source code mean that the code needs to be compiled and the unit tests need to be run many times when doing mutation testing. This makes the process of mutation testing a computationally expensive process. To make mutation testing more viable, various techniques have been developed to reduce the performance cost of mutation testing.

3.1 Mutant reduction

Reducing the number of mutants is a straightforward way of improving the performance of mutation testing. A simple approach, proposed by Acree [1] and Budd [2], is to select a random subset from the set of mutants to use for mutation testing. They showed that using 10% of all possible mutants resulted in only a 16% decline in effectiveness of the mutation testing.
Another approach tries to determine which mutants within the set are redundant by using a method called mutant clustering [4]. Mutants are grouped together using clustering algorithms like k-means and only a small number of mutants from each cluster is selected to be used for mutation testing. This greatly reduces the number of mutants while maintaining a high effectiveness.
It is also possible to reduce the number of mutants by omitting certain mutation operators. It has been shown, for example in the case of the 22 Mothra operators, two of the operators create between 30% and 40% of the mutants [8]. Omitting these two operators severely reduces the number of mutants without having a large impact on the effectiveness of the mutation test.
Using higher order mutants is also suggested as being a good technique to reduce the number of mutants [5]. Using HOMs it is possible to combine multiple FOMs into a single mutant without impacting effectiveness, however, constructing suitable HOMs is still an area which requires more research.

3.2 Execution cost reduction

Rather than reducing the number of mutants, another way of improving the performance of mutation testing is by improving the execution performance of each individual mutant. The traditional mutation testing proposed by DeMilo is also known as strong mutation testing. It is also possible to perform weak mutation where a program is divided into multiple components [3]. When a mutation is performed, rather than checking the mutant after the execution of the entire program, the mutant is checked immediately after the execution of the component in which the mutant exists.

4 Effectiveness of mutation testing

In order to evaluate the effectiveness of mutation analysis testing one can look at the relationship between real faults and the mutants [7]. In several scenarios there is a coupling between real faults and mutants. These scenarios include the replacement or deletion of the following operators:

a. Conditional Operator

b. Relational Operator

c. Statements

There are however scenarios that require more concrete definitions in order to apply mutant generation. These scenarios can be categorized into 3 categories.

a. Real faults requiring stronger mutation operators

b. Real faults requiring new mutant operators

c. Real faults not coupled to mutants

Although mutant analysis testing is in need of more structure and definitions it is still a valid method for testing in practice. This is indicated by the fact that the detection of mutants is correlated with the detection of real faults and is found to be stronger than the detection of real faults in correlations with statement coverage [7].

5 Open problems

There are two fundamental problems with mutation analysis testing [6]:

a. High computational cost
   A single test set can produce multiple mutants for which the original code needs to be run and checked.

b. Amount of human effort
   This can effectively be divided into to two sub-problems.

   (a) Human Oracle Problem
   The Human Oracle Problem represent the need for a human to manually check the original programs output with the result of each test case.

   (b) Equivalent Mutant Problem
   A Equivalent Mutant is a mutant that cannot be killed. The behavior of the resulting program is therefor equivalent to the original one.

The problems stated above cannot be solved completely but their implications can be reduced as the field advances. The Equivalent Mutant problem remains unresolved. The future application of mutant analysis testing may lie withing avoiding and/or reducing the creation of equivalent mutants. Another use could be within languages that do not produces equivalent mutants [6]. Much of the work in the field has been directed at generating mutant and less directed at killing mutants with test case generation.

References

[1] A. T. Acree Jr. On mutation. Technical report, DTIC Document, 1980.

[2] T. A. Budd. Mutation analysis of program test data. 1980.

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

[4] S. Hussain. Mutation clustering. *Ms. Th., Kings College London, Strand, London*, 2008.

[5] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*, pages 249–258. IEEE, 2008.

[6] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, 37(5):649–678, 2011.

[7] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[8] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software: Practice and Experience*, 21(7):685–718, 1991.

[9] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for java. In *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, pages 352–363. IEEE, 2002.