Integrating Mutation Testing into Agile Processes through Equivalent Mutant Reduction via Differential Symbolic Execution

Mark Anthony Cachia
Department of Computer Science
University of Malta
Msida, Malta
mcac0019@um.edu.mt

Mark Micallef
Department of Computer Science
University of Malta
Msida, Malta
mark.micallef@um.edu.mt

I. Agile Programming

In agile programming, software development is performed in iterations. To ensure the changes are correct, considerable effort is spent writing comprehensive unit tests [2]. Unit tests are the most basic form of testing and is performed on the smallest or smaller set of code [7].These unit tests have multiple purposes, the main one being that of acting as a safety net between product releases. However, the value of testing can be called into question if there is no measure of the quality of unit tests [2]. Code coverage analysis is an automated technique which illustrates which statements are covered by tests [4]. However, high code coverage might still not be good enough as whole branches or paths could still go completely untested which in turn leads to false sense of security [8]. Mutation Testing is a technique designed to successfully and realistically identify whether a test suite is satisfactory. In turn, such tests lead to finding bugs within the code. The technique behind mutation testing involves generating variants of a system by modifying operators (called mutants) and executing tests against them. If the test suite is thorough enough, at least one test should fail against every mutant thus rendering that mutant killed. Unkilled mutants would require investigation and potential modification of the test suite [3].

II. Mutation Analysis

Mutation analysis is a process which determines the effectiveness of a test suite. This is achieved by modifying the source of a program and ensuring that at least one test fails thus ensuring the tests are sensitive to particular source changes. A mutant who is detected by tests is called a killed mutant. In mutation analysis, a large number of mutants are generated. Mutants are programs which have been syntactically altered. Sometimes, such alterations lead to the generation of Equivalent Mutants. Equivalent mutants is a problem in Mutation Testing and can be defined as the mutants "which produce the same output as the original program" [3]. Performing Mutation Analysis on Equivalent mutants is a waste of computation time. As program equivalence is undecidable, automatically detecting equivalent mutants is impossible. The equivalent mutant problem is a barrier that prevents Mutation Testing from being widely adopted. The result of Mutation Analysis is a ratio or percentage of the killed mutations divided by the sum of equivalent mutants [3]; Figure 1 illustrates.

$$MS = \frac{\sum KilledMutants}{\sum(Mutants - EquivalentMutants)}$$

Fig. 1. Equation representing the Mutation Score [3]

There are various reasons why mutants may be equivalent. Grun et al. [1] manually investigated eight equivalent mutants generated from the JAXEN XPATH query engine program. They noticed four main reasons which cause a mutant to be equivalent are

- mutants generated from unneeded code,
- mutants which improves speed,
- mutants which just alter the internal states, and
- mutants which cannot be triggered.

III. Symbolic Execution

Symbolic execution is the process of analysing a program by executing it in terms of a symbolic parameter α instead of a concrete instance. The class of inputs represented by each symbolic execution is determined by the control flow of the program based on branching and operations on the inputs. Each branch leads to a unique path condition (PC). A PC is a set of conditions which the concrete variables have to adhere to for the execution to be in the given particular path; hence, for any given set of concrete parameters, the given parameters can reside into at most one path. Initially, the path condition is true, however at each branch operation, the branch condition is ANDed to the previous PC. In the else path, the NOT of the branch condition is added to the current PC [5].

Symbolic execution takes normal execution semantics for granted and is thus considered to be a natural extension of concrete execution. At the end of symbolic execution, an execution tree can be graphed illustrating all the possible paths the program can follow. At each leaf, there exists a unique PC which can be satisfied by a concrete input. The PC at any of the leaves is distinct i.e. $\neg$ ($\mathrm{PC}_1 \bigwedge \mathrm{PC}_n$). As symbolic execution satisfies a commutative property (relative to concrete examples, symbolic execution has the "exact same effects as conventional executions" [5].

A. Differential Symbolic Execution

Differential symbolic execution is a technique introduced by Person et al. [6] which efficiently performs Symbolic execution on two versions of the same class. The aim of Differential symbolic execution is effectively and efficiently determine whether the two version of the code is equivalent; and if not, characterise the behavioural differences by identifying the sets of inputs causing different behaviour [6].

Differential symbolic executions works by symbolically executing each method version to generate symbolic summaries. Symbolic summaries pair input values with the constraint (also known as the branch condition) together with the operation on a symbolic value (the effect or operation). The summaries from both versions are compared. If the summaries are equivalent, the methods are considered functionally equivalent. Otherwise, a behavioural delta (Δ) which characterise the input values where the versions differ.

IV. Proposed work and evaluation

The proposed work is to combine Differential symbolic execution to the most effective optimisations of Mutation Testing. Direct bytecode manipulation as well as Localised Mutation; a technique which only mutates modified code when compared to previous versions as introduced in the FYP, will be employed to ensure the utmost performance in the generation of mutants. Differential symbolic execution, will be used to perform efficient functional equivalence between the mutated method and the original method to ensure the method's semantics have been altered. This determines if the versions have the same black box behaviour as well as have the same partition effects [6].

The evaluation under consideration will asses the effectiveness of equivalent mutant reduction in Mutation analysis. The evaluation will be performed in two iterations with two different algorithms;

1) Mutation testing with the most efficient optimisations
2) Mutations testing with the most efficient optimisations as well as equivalent mutant reduction using Differential symbolic execution.

Various metrics are envisaged to be utilised. Such metrics currently include the execution time to perform the complete Mutation analysis and the time taken for the developer to implement enough tests until a satisfactory Mutation score is achieved. Another aim of this work is to find the most efficient point in the software life cycle at which it is ideal to start performing mutation analysis. Particularly, if it's worth performing in just one chunk at the very end of the cycle or if it is most feasible to perform incremental Mutation testing during the whole development process. It is planned that the interval at which Mutation analysis is ideal to commence and at what interval the analysis should be performed (such as per commit or by some other heuristic) will be also determined during the empirical evaluation.

References

[1] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The impact of equivalent mutants. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops*, ICSTW '09, pages 192–199, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John G. Cleary, Stuart Inglis, and Mark Utting. Jumble java byte code to measure the effectiveness of unit tests. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, TAICPART-MUTATION '07, pages 169–175, Washington, DC, USA, 2007. IEEE Computer Society.

[3] Harman M Jia Y. An analysis and survey of the development of mutation testing. *ACM SIGSOFT Software Engineering Notes*, 1993.

[4] Nguyen H Kaner C, Falk J. *Testing computer software*. 1999.

[5] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[6] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Pǎsǎreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[7] R S Pressman. *Software engineering: a practitioner's approach (2nd ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1986.

[8] Ben H. Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set. *Journal of Systems and Software*, page 2009.