Using Symbolic Execution for Equivalent Mutant Detection

Mark Anthony Cachia[1] and Mark Micallef[1]

Department of Computer Science, University of Malta
{mark.a.cachia.09|mark.micalef@um.edu.mt}

Mutation Testing is a fault injection technique used to measure test adequacy score by generating defects (mutations) in a program and checking if its test suite is able to detect such a change. However, this technique suffers from the Equivalent Mutant Problem [3].

Equivalent mutants are mutants which on mutation retain their semantics [3]. Thus, although equivalent mutants are syntactically different, they remain semantically equivalent to the original program [3]. An automated solution which decides equivalence is impossible, as equivalence of non-trivial programs is undecidable [8, 3]. The fact that the Equivalent Mutant Problem is undecidable usually means that human effort is required to decide equivalence [3]. Equivalent mutants are the barrier keeping Mutation Testing from being widely adopted [3]. Moreover, in one study by Irvine et al [2], the average time taken for each manual mutant classification was fifteen minutes.

This work explores the application of Symbolic Execution to the Equivalent Mutant Problem. Symbolic Execution is a technique which enumerates a program's paths and for each path outputs the conditions variables must satisfy for execution to reside in the path called *Path Conditions* together with the output of the program at that path called *Effects*. The combination of Path Conditionsand Effects are known as the Path Condition ad Effect pair or PCE. A Path Condition is an expression made up of symbolic values representing the conditions the variables must satisfy for execution to reside in a particular path [6, **?**]. If at some point of execution the Path Condition resolves to *false*, i.e. a path condition whose expression cannot be satisfied, the Path Condition is considered to be infeasible as it is mathematically impossible for execution to reach the path. Such PCEs are eliminated on the basis that it is impossible for the program to reach these paths [6].

Symbolic Execution can be used to approximate equivalence. This can be achieved by performing Symbolic Execution on versions of the same program and equating the outputted PCE pairs. The two most applicable variants of Symbolic Execution Differential Symbolic Execution[4] and Directed Incremental Symbolic Execution [5] were analysed however they were deemed to be not efficient enough for the Equivalent Mutant Problem. Hence, an algorithm called SEEM which is both efficient and effective to detect equivalent mutants was to be developed. SEEM works as follows. Initially the PCE pairs of the original version of the *method* being mutated are generated. The main reason why only the mutated method's summary is generated is that as the code executed before the mutation has not been changed. Assuming there is no interleaving, determinism states that

both the original program and the mutated program will execute identically until before the mutated method is called.

After the summary of the original method has been generated, the mutation is performed and the PCE pairs of the mutated method is then generated. However, this process is done intelligently in order to improve efficiency. This is achieved by retracing the coverage of the original program on the mutant. That is, if a mutation is performed on the *then* branch of an *if* statement, only the paths passing through the if statement is explored. If the PCE pairs of the original program passing through the if statement are equal to the PCEs of the mutant, the mutant is considered to be equivalent. A proof of concept tool was implemented which makes use of the Microsoft Z3 constraint solver [1] used to determine feasibility and simplify Path Conditions.

Three separate investigations were performed to evaluate the *Effectiveness*, *Efficiency* and the extent to which the tool handles different levels of complexities in the Path-Explosion Problem. Various scenarios leading to equivalent mutants were encountered in the course of the work in which SEEM was able to correctly classify all but one. SEEM was compared to the state of the art tool Javalanche which performs Equivalent Mutant Detection by invariant violations [7]. When SEEM was used to classify the same mutants as Javalanche, SEEM was 89% accurate whilst Javalanche was only 65% accurate. The theoretical efficiency of SEEM was also studied. It was determined that the time and space saved by employing SEEM instead of traditional is $SEEM\ time\ savings = O(n^{\sum program\ branches\ -\ \sum method\ branches})$. The final experiment was conducted in order to obtain typical running times of SEEM. Several experiments were performed. The highest running time recorded was that of just over nine seconds in a mutant which had over a thousand paths to explore.

From the results achieved, it was concluded that SEEM is a suitable technique to be used in the reduction of the Equivalent Mutant Problem.

References

1. Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.
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. Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. 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.

5. Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.
6. Corina S. Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, October 2009.
7. David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 69–80, July 2009.
8. Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.