Making mutation testing a more feasible proposition for the industry

Mark Micallef, Mark Anthony Cachia

Deptartment of Computer Science, University of Malta, Malta
mark.micallef@um.edu.mt, mcac0019@um.edu.mt

Software engineering firms find themselves developing systems for customers whose need to compete often leads to situations whereby requirements are vague and/or prone to change. One of the prevalent ways with which the industry deals with this situation is through the adoption of so-called Agile development processes. Such processes enable the evolutionary delivery of software systems in small increments, frequent customer feedback, and, ultimately, software which continuously adapts to changing requirements. In this fluid scenario, developers rely on automated unit tests to gain confidence that any regressions resulting from code changes will be detected. Consequently, trust in the software system can only follow from the quality of the tests. Unfortunately, the industry tends to rely on tools that calculate primitive measures such as statement coverage, a measure which has been shown to provide a false sense of security [2].

Mutation testing [3] is an analysis technique based on the following idea: Given a program P and a test suite T, if one judges T to adequately cover P, then executing T against P' (where P' is an altered version of P), should result in at least one failing test. Therefore, from an original program P, a number of modified programs $P^1 ... P^n$, called *mutants*, are produced by applying a number of syntactic modifications to P. These modifications are carried out by mutation operators which are designed to change P in a way that corresponds to a fault which could be introduced by a developer. Mutants which are detected by T are said to be *killed*. Undetected (*unkilled*) mutants require manual investigation by developers, possibly resulting in improvements to T. In comparison to techniques such as statement coverage analysis, mutation testing provides a significantly more reliable measure of test suite thoroughness.

Despite its effectiveness, mutation testing suffers from three recognised problems. These are (1) the computational expense of generating mutants[1], (2) the time required to execute test suites against all mutants[2], and (3) the equivalent mutant problem. The latter refers to situations where syntactically different mutants turn out to be semantically identical, thus wasting time and effort. Besides the three cited problems with mutation testing, we also argue that there

[1] The computational complexity of Mutation Testing is $O(n^2)$ where n is the number of operations in the source code [1].
[2] Executing all tests against each mutant renders mutation tools unacceptably slow and only suitable for testing relatively small programs[4].

is a fourth problem, one concerned with the time and effort required to investigate and address unkilled mutants. Each unkilled mutant requires a developer to understand the mutant's semantics, determine if a change to the test suite is required and finally modify the test suite to kill the mutant. We argue that this effort can be a deterrent to the wider uptake of mutation testing because the time and cognitive effort required to carry out the task may not be perceived as being worth the potential benefits gained.

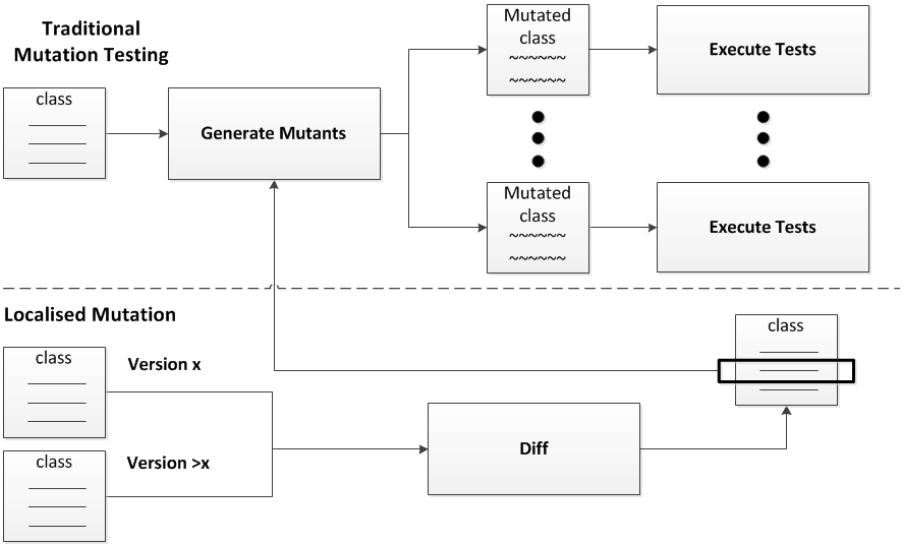

Fig. 1. Comparing traditional mutation testing with localised mutation

In this talk, we will provide an overview of mutation testing and subsequently discuss problems which prevent its wider uptake. We then discuss our research activities in this area and present a technique which we term as *localised mutation*. The technique leverages the iterative nature of Agile development such that one only creates mutants from sections of the codebase which have changed since the last mutation run. The hypothesis is that if mutation testing is carried out in bite-sized chunks on code which has recently changed, then computational expense can be drastically reduced and developers should experience less cognitive load during analysis. Consequently, the main hurdles of mutation testing adoption in industry would be significantly reduced. Preliminary results from this research will also be discussed and related to our ongoing research activities.

References

1. M. E. Delamaro, J. Maldonado, A. Pasquini, and A. P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. *Empirical Softw. Engg.*, 6(2):111–

142, June 2001.
2. S. A. Irvine, T. Pavlinic, L. Trigg, J. G. Cleary, S. Inglis, and M. 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. H. M. Jia Y. An analysis and survey of the development of mutation testing. *ACM SIGSOFT Software Engineering Notes*, 1993.
4. E. W. Krauser, A. P. Mathur, and V. J. Rego. High performance software testing on simd machines. *IEEE Trans. Softw. Eng.*, 17(5):403–423, May 1991.