Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science 294 (2013) 2–11

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Towards Incremental Mutation Testing

Mark Anthony Cachia[1,2] Mark Micallef [3] Christian Colombo [4]

Department of Computer Science
University of Malta
Msida, Malta

Abstract

Proponents of Agile development processes claim that adhering to Agile principles leads to the delivery of high quality code in evolutionary increments. Confidence in resulting systems is mostly gained through the use of unit test suites, entrusted to catch regressions as soon as they occur. Consequently, the system can only be as trustworthy as its tests, meaning that measurements of the tests' quality is crucial. Whilst mutation testing has been proposed as a means of uncovering test suite defects, it has not been widely adopted in the industry; mainly due to its computational expense and manual effort required by developers investigating unkilled mutants. To make mutation testing affordable, we propose *incremental mutation testing* — a variation of mutation testing which leverages the iterative nature of agile development by limiting the scope of mutant generation to sections of code which have changed since the last mutation run. Preliminary results show that the number of mutants generated is drastically reduced along with the time required to generate mutants and execute tests against them.

Keywords: Mutation Testing, Agile Development Processes, Incremental Mutation Testing

1 Introduction

Software engineering firms find themselves developing systems for customers whose need to compete often leads to situations whereby requirements are vague and 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 perceived quality of the tests. Unfortunately, the industry tends to rely on tools

[1] The authors would like to thank Gordon Pace and the reviewers for their useful feedback.

[2] Email: mcac0019@um.edu.mt

[3] Email: mark.micallef@um.edu.mt

[4] Email: christian.colombo@um.edu.mt

doi:10.1016/j.entcs.2013.02.012

that calculate primitive measures such as statement coverage; a measure which has been shown to provide a false sense of security [1].

Mutation testing [2] is an analysis technique which systematically creates faulty versions of a program (called mutants) and checks whether the program's test suite detects the fault. If a particular mutant goes undetected, i.e. no tests fail against the mutant, it is said to be *unkilled* and flagged for investigation by a developer. Although mutation testing is an effective technique for measuring a test suite's thoroughness, it has not found its place in the industry. Two main reasons for this is due to (1) the computational expense incurred when generating/killing mutants — meaning that significant amount of time and resources would have to be dedicated to mutation testing, and (2) the length of time elapsed from the development time till the developers receive feedback — meaning that the developers would find it more difficult to act upon it.

The contribution of this paper is a technique which we term as *incremental mutation testing*. The technique leverages the evolutionary nature of Agile development whereby developers are committed to ongoing improvement of a product in small regular increments throughout its lifetime. Our technique leverages this by applying mutation testing in a similar manner as a system evolves. If we start with a fully-tested codebase (initially an empty codebase), then mutation testing need only be carried out on sections of the code which are affected by changes as the system evolves. We prove the soundness of this approach and show that if mutation testing is performed incrementally, the computational expense can be drastically reduced and as a result developers would benefit from short feedback loops, facilitating their analysis. Consequently, the main hurdles of mutation testing adoption in industry would be significantly reduced.

The rest of this paper is organised as follows. Section 2 briefly covers the principles behind mutation testing and discusses problems preventing its wider adoption. This is followed by Section 3 which provides an overview of incremental mutation testing and formally shows the sanity of the approach. Next, in Section 4, we give an instantiation of incremental mutation testing and present a preliminary evaluation of the idea. Finally, Section 5 provides an overview of related work in the literature whilst Section 6 draws conclusions and discusses our future plans in this area.

2 Background

Mutation testing [2] (depicted in Figure 1[top]) is a technique which analyses the thoroughness of a test suite using fault injection. In essence, given a program P and a test suite T which tests P, the approach involves generating faulty variations of P (called mutants) and checking whether for every mutant, there is at least one test case in T which fails. We write $T(P)$ to denote a successful run of test suite T on program P and $\neg T(P)$ to denote that at least one of the tests in the test suite has failed on P.

Mutation testing begins by generating a set of programs $P_1, P_2, \ldots, P_n$ using a set of mutation operators represented by the function $\mathcal{M}$ on the program P,

$\mathcal{M}(P) = \{P_1, P_2, \ldots, P_n\}$. These programs are usually syntactically similar to P but never (syntactically) equivalent to it. That is to say $\forall i : 1..n \cdot P_i \not\equiv P$. Although there are an infinite number of possible mutants, mutation operators found in the literature usually produce mutants by applying deterministic transformation rules such as '+' $\rightarrow$ {'−','×' ,'÷' }. In this case, for every instance of '+' in a program, three mutants will be generated, each with '+' replaced by '−','×' and '÷' respectively. This results in a quadratic computational complexity based on the number of operations involved in the mutation operators and their frequency in the source code [2].

T is said to *cover* P, i.e. T adequately tests P, if executing T against any $P_i \in \mathcal{M}(P)$ results in at least one failing test. In such cases we say that the mutant P_i is *killed* by T. If on the other hand, no test failures occur, we state that P_i is an *unkilled* mutant which might indicate that T does not in fact cover P. In such cases, a manual investigation is required to establish why P_i was not killed.

Definition 2.1 A test suite T is said to cover a program P, denoted $T \triangleright P$ if and only if P satisfies T, $T(P)$, while any $P_i \in \mathcal{M}(P)$ fails the test suite, $\neg T(P_i)$:

$$T \triangleright P \stackrel{\text{def}}{=} T(P) \wedge \forall P_i \in \mathcal{M}(P) \cdot \neg T(P_i)$$

The ratio of killed mutants to total mutants is known as the mutation score and provides a measure of test suite coverage in the context of the generated mutants. Mutation operators are usually designed to change P in a way that corresponds to a fault which could be introduced by a developer. Consequently, in comparison to techniques such as statement coverage analysis, mutation testing provides a significantly more reliable measure of test suite thoroughness [3, 4]. Despite its effectiveness, mutation testing suffers from three recognised problems [2]. Firstly, whilst the polynomial computational complexity of mutation testing does not seem prohibitive, in a typical commercial system the large amount of potential mutation points would make the computational expense considerably high. Secondly, once mutants have been generated, each one needs to be tested against the original program's test suite. Considering that test suites on large systems will optimistically take a few minutes to execute, the time required for this task would be considerable. The third cited problem with mutation testing is the so-called *equivalent mutant problem* whereby syntactically different mutants turn out to be semantically identical, thus wasting time and effort. Besides these three cited problems, we also argue that there 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.

In this work, we focus on the first two problems — both contributing to the computational expensiveness of mutation testing — by presenting an incremental approach to mutation testing. This approach generates mutants only for the points

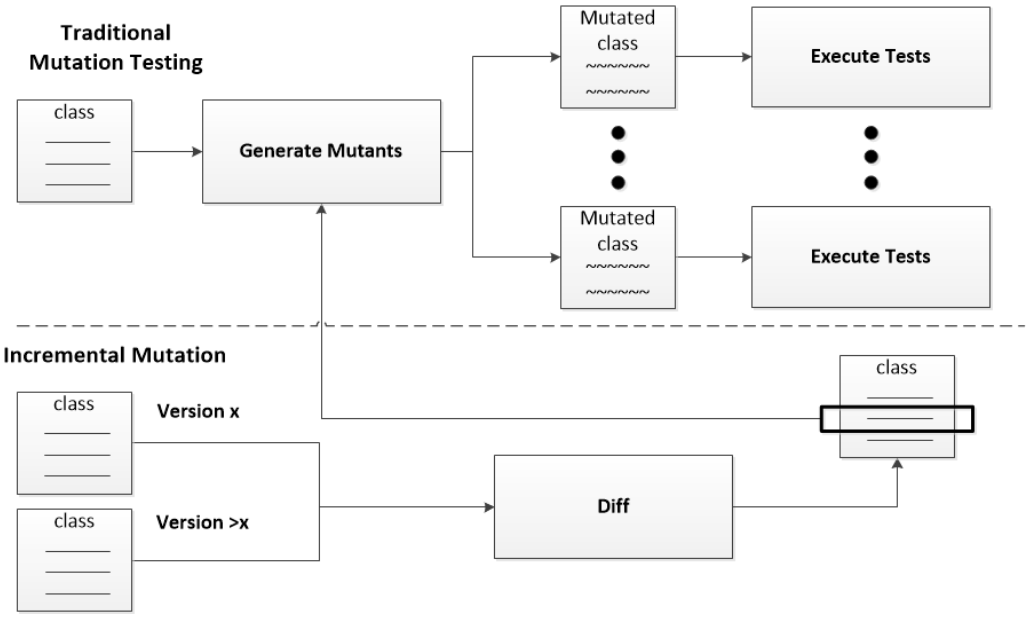

Fig. 1. Comparing traditional mutation testing with incremental mutation testing

in the program which have changed since the last application of mutation testing — tackling the first problem. Furthermore, it only invokes the part of the test suite which tests the changed part — tackling the second problem. Indirectly, this also improves the feedback loop to the developers since the feedback they get is focused on the parts which they have been working on recently.

3 Incremental Mutation Testing

Incremental mutation testing attempts to alleviate problems associated with mutation testing by leveraging the evolutionary and test-driven nature of Agile development. The underpinning idea is that of limiting the scope of mutation testing to code that has changed [5] within the context of two particular versions of code. By applying mutation testing on each change across successive versions of the code, over the entire evolutionary process, one would have effectively applied mutation testing over the whole system, incrementally. More precisely, incremental mutation testing assumes two programs P_1 and P_2 where P_2 is an evolution of P_1 such that P_2 consists of two parts: a changed part (${P_2}^{\delta}$) which has evolved from a corresponding part of P_1 (${P_1}^{\delta}$), and an unchanged part (${P_1}^{\not\delta} = {P_2}^{\not\delta} = P^{\not\delta}$) with respect to P_1. We therefore represent P_1 and P_2 as $P_1 = {P_1}^{\delta} + P^{\not\delta}$ and $P_2 = {P_2}^{\delta} + P^{\not\delta}$. In this context, the composition operator + assumes that there is a way of splitting a program up into two parts such that the parts can be tested independently [6]. Similarly, we assume that there is a way of splitting the test suite into (potentially overlapping) parts which test the corresponding program parts. Formally, we assume that for

[5] Unless otherwise specified, references to code *changes* in this paper refer to syntactic changes.

[6] In practise, this can be generally realised through the use of static analysis.

any program $P = P^{\delta} + P^{\not\delta}$ and test suite $T = T^{\delta} + T^{\not\delta}$:

$$T(P) \iff T^{\delta}(P^{\delta}) \wedge T^{\not\delta}(P^{\not\delta}) \qquad (\dagger)$$

Furthermore, we assume that given the composition of two programs, the mutation operator applies itself on each part in turn, i.e. it is never the case that the change spans across the two parts. This assumption enables us to reason about the sub-parts independently in the proofs which follow. Note that simple mutants, i.e. mutants which change the program through one mutation operator at a time, naturally satisfy this assumption. More formally:

$$\mathcal{M}(P + P') = \{\forall P_i \in \mathcal{M}(P) \cdot P_i + P'\} \cup \{\forall P'_i \in \mathcal{M}(P') \cdot P + P'_i\} \qquad (\ddagger)$$

Proposition 3.1 *If a test suite T covers P where T can be split into $T^{\delta} + T^{\not\delta}$ and correspondingly $P = P^{\delta} + P^{\not\delta}$, then the split parts of the test suite cover P^{δ} and $P^{\not\delta}$ individually:*

$$T \triangleright P \iff T^{\delta} \triangleright P^{\delta} \wedge T^{\not\delta} \triangleright P^{\not\delta}$$

Proof.

$$\begin{aligned}
& T \triangleright (P^{\delta} + P^{\not\delta}) \\
& \{\textit{By definition of } \triangleright\} \\
\iff & T(P^{\delta} + P^{\not\delta}) \wedge \forall P_i \in \mathcal{M}(P^{\delta} + P^{\not\delta}) \cdot \neg T(P_i) \\
& \{\textit{By definition of } \ddagger\} \\
\iff & T(P^{\delta} + P^{\not\delta}) \wedge (\forall {P_i}^{\delta} \in \mathcal{M}(P^{\delta}) \cdot \neg T({P_i}^{\delta} + P^{\not\delta})) \\
& \qquad\qquad \wedge (\forall {P_i}^{\not\delta} \in \mathcal{M}(P^{\not\delta}) \cdot \neg T(P^{\delta} + {P_i}^{\not\delta})) \\
& \{\textit{By } \dagger \textit{ trice and de Morgan's Law}\} \\
\iff & T^{\delta}(P^{\delta}) \wedge T^{\not\delta}(P^{\not\delta}) \wedge (\forall {P_i}^{\delta} \in \mathcal{M}(P^{\delta}) \cdot \neg T^{\delta}({P_i}^{\delta}) \vee \neg T^{\not\delta}(P^{\not\delta})) \\
& \qquad\qquad \wedge (\forall {P_i}^{\not\delta} \in \mathcal{M}(P^{\not\delta}) \cdot \neg T^{\delta}(P^{\delta}) \vee \neg T^{\not\delta}({P_i}^{\not\delta})) \\
& \{\textit{By predicate logic}\} \\
\iff & T^{\delta}(P^{\delta}) \wedge T^{\not\delta}(P^{\not\delta}) \wedge (\forall {P_i}^{\delta} \in \mathcal{M}(P^{\delta}) \cdot \neg T^{\delta}({P_i}^{\delta})) \\
& \qquad\qquad \wedge (\forall {P_i}^{\not\delta} \in \mathcal{M}(P^{\not\delta}) \cdot \neg T^{\not\delta}({P_i}^{\not\delta})) \\
& \{\textit{By definition of } \triangleright\} \\
\iff & T^{\delta} \triangleright P^{\delta} \wedge T^{\not\delta} \triangleright P^{\not\delta}
\end{aligned}$$

□

Given that a test suite has been shown to adequately cover a system under test, in the following evolution of the code this information can be used to minimise the number of mutations required to check the test suite. Intuitively, this is achieved by eliminating the unchanged part of the system from mutation testing: if the second version of the code can be split into the changed part and the unchanged part, incremental mutation testing assumes that tests relating to the unchanged part do not need to be analysed for thoroughness because this would have been done in previous evolutions of the code. More formally, this idea is captured in the following

theorem.

Theorem 3.2 *If the system code $P_1 = {P_1}^{\delta} + P^{\not\delta}$ has been shown to be adequately covered by a test suite T_1, $T_1 \triangleright ({P_1}^{\delta} + P^{\not\delta})$, then to show that the new version is also adequately covered, $T_2 \triangleright ({P_2}^{\delta} + P^{\not\delta})$, it suffices to check that ${T_2}^{\delta} \triangleright {P_2}^{\delta}$:*

$$T_1 \triangleright ({P_1}^{\delta} + P^{\not\delta}) \wedge {T_2}^{\delta} \triangleright {P_2}^{\delta} \implies T_2 \triangleright ({P_2}^{\delta} + P^{\not\delta})$$

Proof.

$$\begin{aligned}
& T_1 \triangleright ({P_1}^{\delta} + P^{\not\delta}) \wedge {T_2}^{\delta} \triangleright {P_2}^{\delta} \\
& \{\textit{By Proposition 3.1}\} \\
\implies & {T_1}^{\delta} \triangleright {P_1}^{\delta} \wedge {T_1}^{\not\delta} \triangleright P^{\not\delta} \wedge {T_2}^{\delta} \triangleright {P_2}^{\delta} \\
& \{\textit{By propositional logic, Proposition 3.1 and } {T_1}^{\not\delta} = {T_2}^{\not\delta}\} \\
\implies & T_2 \triangleright ({P_2}^{\delta} + P^{\not\delta})
\end{aligned}$$

□

In the next section we present an instantiation of incremental mutation testing based on the above theory and show its applicability to a real-life case study.

4 Instantiation

Any implementation of incremental mutation testing assumes two fundamental properties of the underlying framework: (1) that the code can be split into the changed and the unchanged parts; and (2) that the test suite can also be split into a part which tests the changed part and a part which tests its counterpart. To facilitate this process, in our instantiation of incremental mutation testing, we choose *methods* as our smallest unit of consideration. This particularly makes sense in the context of unit testing where typically a unit test tests a method (rather than for example a single statement). Thus, as regards to splitting the system into changed and unchanged parts, we consider any method with a change (even if it is just for a single statement) to be part of the changed part of the system and vice-versa. Once we identify all the changed methods, using static analysis we delineate all the unit tests which invoke any of the changed methods (similar to Schuller and Zeller's [5] work in Javalanche). Together, these two aspects give us an instantiation of incremental mutation testing (depicted in Figure 1).

Admittedly, this instantiation is naïve since it does not consider the interconnections across methods with two consequences: (1) methods which depend on the changed methods are not included for mutation generation, and consequently (2) tests which check methods affected by the change are not included for killing mutations. Notwithstanding these limitations, we applied our approach to a real-life case study with promising results.

4.1 Evaluation Process

In order to observe the technique in different scenarios, the evaluation took the form of an experiment on three scenarios pulled of a candidate codebase. Each scenario

consisted of executing a purposely build mutation testing tool [7] on two versions of the source code which were purposely selected to represent an archetype of a development cycle. The archetypes represented development cycles which contained high, medium and low code churn. These scenarios are outlined in Table 1. Besides the level of code churn, two variables were identified as having the potential of influencing the experiment, the first of which was the selection of mutation operators. The decision in this regard was based on the *Mutation Coupling Effect Hypothesis* which states that complex mutants are coupled to simple mutants in such a way that a test data set that detects all simple mutants in a program will also detect a large percentage of the complex mutants [6]. As a result, it was decided that only seventeen commonly used simple mutant operators would be used during the experiment. The second variable considered was the choice of a candidate codebase. Due to this being only a preliminary evaluation, the Apache Commons CLI Library, consisting of a modest yet non-trivial 5 KLOC, was selected for the task. Apart from being open-source, the CLI library makes limited use of object-oriented constructs and thus fits our selection of simple mutation operators. Furthermore, the codebase comes with a unit test suite that boasts 97% statement coverage signifying a mature test suite for which one does not expect to find an excessive number of unkilled mutants.

The experiment involved carrying out traditional mutation testing and incremental mutation testing on all three scenarios. Considering that each scenario consisted of two versions of the code v_1 and v_2, with v_2 occurring chronologically after v_1, then traditional mutation testing was carried out on v_2 for each scenario while incremental mutation testing was applied on the the differences between v_1 and v_2. In each case, we collected data about the total number of generated mutants, the number of killed mutants and the execution time of the end-to-end process including static analysis to select which tests to execute.

4.2 Results

The data collected during the experiment allowed us to compare and contrast the characteristics of incremental mutation testing to traditional mutation testing. The results, which are summarised in Table 2, confirm that incremental mutation testing significantly reduces the amount of mutants generated since in Scenarios 1, 2, and 3 incremental mutation testing generated 91%, 62% and 46% less mutants respectively. The results also indicate that the smaller the code churn between the two versions of code being compared, the less mutants are generated — resulting in faster execution time. In fact, execution time is significantly decreased through the use of incremental mutation testing such that speed improvements of between 88% and 91% were observed.

While these results are encouraging from the point of view of computational expense and the consequent timely feedback, the results are less clearcut when it comes to the number of unkilled mutants. As expected the kill rate drops from

[7] The tool can be downloaded from http://www.um.edu.mt/__data/assets/file/0007/175957/IncrementalMutation_v0_1.zip

#	Scenario	LOC Affected	# Methods Affected
1	Low code churn	12	1
2	Medium code churn	60	4
3	High code churn	720	24

Table 1
The three scenarios considered during evaluation

Scenario	Total Mutants	Δ	Unkilled Mutants	Kill Rate	Time (s)	Δ
1 - MT	349	91%	35	90%	58	91%
1 - IMT	30		5	83%	5	
2 - MT	253	62%	15	94%	42	88%
2 - IMT	95		51	46%	5	
3 - MT	340	46%	97	71%	79	89%
3 - IMT	183		126	31%	9	

Table 2
Comparison of mutant generation, mutant killing and execution time using both traditional (MT) and incremental (IMT) mutation testing for each scenario

mutation testing to incremental mutation testing since we are focusing on the part which typically has more problems in the test suite. However, the number of unkilled mutants increases in Scenarios 2 and 3 because of the naïve way in which we are selecting the tests to execute — further experimentation (not shown in the table below) showed that executing more tests results in more mutants being killed. Another notable phenomenon which occurred in Scenario 1 is that the number of generated mutants in the incremental approach is less than the number of unkilled mutants of traditional mutation testing. This means that a significant number of unkilled mutants lie outside of the code section delineated by our approach. We believe that this is due to the simplistic approach in delineating the code which has been affected by the code changes but we leave this issue for future investigation.

5 Related Work

Within the field of mutation testing, various attempts have been made to optimise and reduce the computational cost of the technique including: *selective mutation* whereby mutation operators are strategically selected [7], *higher order mutation* in which multiple mutations are combined into individual mutants [8], and Schuller and Zeller's [5] approach to only execute tests which exercise mutated sections of the code. While these techniques are complementary to ours (indeed we include concepts from Schuller and Zeller's work in incremental mutation and can easily

integrate other cited optimisations with our technique), we mainly draw our inspiration for incremental mutation testing from other areas in software engineering — to the best of our knowledge an incremental approach has never been proposed for mutation testing.

JUnitMax [8] is a unit test runner which was designed with the goal of providing ongoing feedback to developers while they work. As a developer writes code and saves it, JUnitMax automatically checks which part of the code has changed and non-intrusively executes relevant test cases in the background. If there are any failures, the developer is notified. This significantly shortens feedback loops and also leads to faster fix times because failures are likely to be related to something which the developer has just done. This is similar to incremental mutation testing in that the aim is to provide regular bite-sized feedback about the quality of a test suite as code evolves. Symbolic execution [9] and automated static code analysis [10] are both useful techniques which like mutation testing suffer from scalability issues. Attempts to address this problem have leveraged the incremental nature of software development to perform symbolic execution efficiently [11,12] and to selectively display results of automated static code which developers are likely to be interested in [10]. While incremental mutation testing is not directly related to these fields, we combine these ideas to optimise the computational efficiency of mutation testing and shorten the feedback loop to the developers.

6 Conclusion and Future Work

Evolutionary-based software development processes highly depend on their supporting test suites to ensure no regressions occur from one evolution to the next. With this reliance on tests to ensure the reliability of software systems, one cannot help but find means of ensuring the quality of the tests — their coverage. Whilst mutation testing has been shown to be effective in discovering defects in test suites, it has still not been widely adopted mainly due to the overhead it represents in computation as well as the time it takes for feedback to reach developers.

In this paper we introduced incremental mutation testing, a variation of mutation testing which is applied incrementally across the evolutions of a software development life cycle. We have formally defined the concept and shown it to be sound. Moreover, we have instantiated incremental mutation testing and applied it to a modest case study whose preliminary evaluation indicates that the technique alleviates the problems of prohibitive computational expense and timely feedback to the developers. Although the results are promising, the evidence shows that our current instantiation of incremental mutation testing is simplistic in the way it localises mutation testing. In the future we aim to apply more intelligent approaches for test case selection so as to take into account the relationships across system units. Furthermore, we also plan to continue towards reducing the cognitive overload issue by integrating incremental mutation testing within the software development process (á la JUnitMax) providing even more timely feedback to de-

[8] http://www.junitmax.org

velopers while also automatically eliminating any unkilled mutants which point to a defect in the test suite which has just been fixed.

This work fits within our overarching aim of making mutation testing feasible in industrial settings. It is hoped that this line of work will lead to a situation where mutation testing will indeed become commonplace in commercial development scenarios, thus allowing companies to reap the benefits of this powerful analysis technique.

References

[1] 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.

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

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 402–411, New York, NY, USA, 2005. ACM.

[4] Márcio Eduardo Delamaro, JoséCarlos Maldonado, Alberto Pasquini, and Aditya P. Mathur. Interface mutation test adequacy criterion: An empirical evaluation. *Empirical Softw. Engg.*, 6(2):111–142, June 2001.

[5] David Schuler and Andreas Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 297–298, New York, NY, USA, 2009. ACM.

[6] A. Offutt. The coupling effect: fact or fiction. In *Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification*, TAV3, pages 131–140, New York, NY, USA, 1989. ACM.

[7] A. Jefferson Offutt and Ronald H. Untch. Mutation testing for the new century. chapter Mutation 2000: uniting the orthogonal, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[8] Yue Jia and Mark Harman. Higher order mutation testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009. ¡ce:title¿Source Code Analysis and Manipulation, SCAM 2008¡/ce:title¿.

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

[10] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Information and Software Technology*, 53(4):363–387, April 2011.

[11] 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.

[12] 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.