Science of Computer Programming 78 (2013) 345–363

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Semantic mutation testing

John A. Clark [a], Haitao Dan [b], Robert M. Hierons [b,*]

[a] Department of Computer Science, University of York, York, YO10 5GH, United Kingdom
[b] School of Information Systems, Computing and Mathematics, Brunel University, Uxbridge, Middlesex, UB8 3PH, United Kingdom

ARTICLE INFO

Article history:
Available online 20 April 2011

Keywords:
Mutation testing
Semantics
Misunderstandings

ABSTRACT

Mutation testing is a powerful and flexible test technique. Traditional mutation testing makes a small change to the syntax of a description (usually a program) in order to create a mutant. A test suite is considered to be good if it distinguishes between the original description and all of the (functionally non-equivalent) mutants. These mutants can be seen as representing potential small slips and thus mutation testing aims to produce a test suite that is good at finding such slips. It has also been argued that a test suite that finds such small changes is likely to find larger changes. This paper describes a new approach to mutation testing, called semantic mutation testing. Rather than mutate the description, semantic mutation testing mutates the *semantics* of the language in which the description is written. The mutations of the semantics of the language represent possible *misunderstandings* of the description language and thus capture a different class of faults. Since the likely misunderstandings are highly context dependent, this context should be used to determine which semantic mutants should be produced. The approach is illustrated through examples with statecharts and C code. The paper also describes a semantic mutation testing tool for C and the results of experiments that investigated the nature of some semantic mutation operators for C.

1. Introduction

Testing is an important but expensive part of the software development process, often consisting of in the order of fifty percent of the overall development budget. Test automation has the potential to make testing more efficient and effective and thus to lead to cheaper, higher quality software.

Mutation testing is an approach to test automation that aims to produce test cases that are good at distinguishing between some description N and variants of it. Each variant is produced by applying a mutation operator to N. A test case t *kills* a mutant M of N if it distinguishes between M and N, typically by M and N producing different output when run with t. A mutant M of N is said to be an *equivalent mutant* if no possible test case kills M. In mutation testing, either a test suite is judged against the mutants created (by determining what percentage of non-equivalent mutants are killed by the test suite) or a test suite is produced to kill all of the non-equivalent mutants. The motivation is that a test suite that is good at distinguishing N from variants of N is likely to be good at finding faults that are similar to applications of the mutation operators.

In traditional mutation testing, the mutation operators are designed to represent syntactically small errors. For example, an operator might replace $+$ by $-$ in an arithmetic expression. In this paper, we propose a different approach to mutation testing, namely semantic mutation testing (SMT). When testing an entity, test generation is based on a model written in some description language (such as a programming language, a design language or a specification language). While many mistakes are slips, other mistakes are the consequence of a misunderstanding of the semantics of the description language. Such misunderstandings may be captured by mutating the semantics of the description language. It is possible to introduce

* Corresponding author.
E-mail addresses: jac@cs.york.ac.uk (J.A. Clark), haitao.dan@brunel.ac.uk (H. Dan), rob.hierons@brunel.ac.uk (R.M. Hierons).

0167-6423/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2011.03.011

changes that reflect *small misunderstandings* regarding the language through making small changes to the semantics of this language. Such changes result in the same model being interpreted in a different way. This contrasts with traditional mutation testing in which small changes are made to the syntax of the model. SMT thus aims to find a different class of fault and should complement traditional mutation testing. When dealing with a programming language, semantic mutation testing can be seen as a process that mutates the compiler rather than the program. Previous work has discussed semantic size in the context of mutation testing [1] but this work did not discuss semantic mutation testing.

This paper makes a number of contributions. First, it introduces semantic mutation, describes its potential role in testing, and explains how it can be implemented. It describes the error model of SMT and several scenarios in which SMT might play a significant role. Examples of semantic misunderstandings for statecharts and the C language are given. We investigate the corresponding mutation operators that reflect differences between semantics, demonstrating that SMT can uncover faults arising from these differences. The differences between SMT and traditional mutation testing are summarised. We then describe a semantic mutation tool that has been developed for C code and several semantic mutation operators that have been implemented. This is followed by results of experiments performed to investigate the nature of the implemented semantic mutation operators and how they compare with related syntactic operators. The description of the tool and the results of the experiments form the main contribution beyond the earlier conference version [2].

The paper is structured as follows. Section 2 describes traditional mutation testing and Section 3 outlines SMT. Section 4 describes the error model of SMT and several scenarios in which SMT might play a significant role. Section 5 then gives examples of situations in which SMT can be applied to statecharts and the C language. Section 6 describes a semantic mutation tool for C. Section 7 describes the experiments and their outcome while Section 8 explores the results and Section 9 discusses threats to validity. Finally Section 10 draws conclusions.

2. Traditional mutation testing

The idea behind mutation testing is simple and intuitively appealing. Mutants are produced by making changes to the program. These changes simulate classes of faults and test cases are produced to distinguish our original program from the mutants. A test suite distinguishing between the original program and the mutants provides confidence in it detecting such classes of faults.

Mutants are produced through the application of mutation operators. Each of these may be applied to a relevant point in a program in order to produce a mutant. The mutation operators involve small syntactic changes. For example, $+$ might be replaced by $-$, $>$ might be replaced by $\geq$, a variable in an expression may be replaced by a constant, or part of an expression may be deleted. The use of such mutation operators is usually justified by the competent programmer hypothesis, which states that competent programmers make small mistakes [3]. There is an issue here—a competent programmer might make semantically small mistakes that cannot be captured by syntactically small changes.

When considering programs, there are several notions as to what it means for a test case t to distinguish between a program N and a mutant M. Under strong mutation testing, which is the original form of mutation testing, M and N are distinguished if they produce a different output on t [3,4]. In weak mutation testing, M and N are distinguished if they produce a different value for some state variable immediately after the point at which N was changed [5]. Firm mutation testing generalises these by allowing the tester to choose the point at which the value of some state variable must differ [6].

The use of only a single mutation operator will often create large numbers of mutants even when the original program is quite small. For this reason, it is normal to restrict the number of mutants produced by using only first-order mutants: those that can be produced from the original program by the single application of one mutation operator. The use of first-order mutants is justified by the coupling hypothesis that states that any test suite that kills all first-order mutants will kill most higher-order mutants. Empirical studies suggest that there is some truth in the coupling hypothesis [7] though many questions still remain. The coupling hypothesis has also been validated by theoretical work [8], although this work makes many assumptions.

Mutation testing was originally applied to programs (see, for example, [9–13,3,14,4,15,16,5,17–20,7,21–26,6]) but more recently it has been applied to other forms of descriptions such as specifications (see, for example, [27–31]). This approach involves producing test cases that kill mutants of the specification, the test cases then being applied to the code. Naturally, in order to do this we need a particular type of specification language — one that can be executed, that can be simulated, or that allows some formal reasoning. In this work we want to produce mutants that are not equivalent. In contrast, some work on applying mutation testing to Communicating Sequential Processes (CSP) specifications considers properties of the specification, and whether these are preserved, and not functional equivalence [32]. Thus, a mutant is killed if it does not satisfy the property of interest. Interestingly, in this context equivalent mutants correspond to fault tolerance and thus their existence is desirable.

Mutation testing has a number of advantages. First, it allows the tester to target particular classes of faults. Should a program pass a test suite that kills all mutants, then it is clear that the non-equivalent mutants produced were not correct. This eliminates a set of faulty behaviors. It also gives us confidence in the test suite distinguishing between a correct program and a program with one of these types of faults. Second, other test criteria may be simulated using mutation testing. Consider, for example, the mutation operator that replaces a statement by a new statement that terminates execution with an error message. Then, any test suite that kills all of the non-equivalent mutants formed using this mutation operator must also provide 100% statement coverage: every reachable statement is executed during testing.