The Journal of Systems and Software 115 (2016) 18–30

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Mutation testing cost reduction by clustering overlapped mutants

Yu-Seung Ma[a,*], Sang-Woon Kim[b]

[a] Electronics and Telecommunications Research Institute, Daejeon, South Korea
[b] FormalWorks, Inc., Seoul, South Korea

ARTICLE INFO

Article history:
Received 20 October 2014
Revised 24 December 2015
Accepted 6 January 2016
Available online 13 January 2016

Keywords:
Software testing
Mutation testing

ABSTRACT

Mutation testing is a powerful but computationally expensive testing technique. Several approaches have been developed to reduce the cost of mutation testing by decreasing the number of mutants to be executed; however, most of these approaches are not as effective as mutation testing which uses a full set of mutants. This paper presents a new approach for executing fewer mutants while retaining nearly the same degree of effectiveness as is produced by mutation testing using a full set of mutants. Our approach dynamically clusters expression-level weakly killed mutants that are expected to produce the same result under a test case; only one mutant from each cluster is fully executed under the test case. We implemented this approach and demonstrated that our approach efficiently reduced the cost of mutation testing without loss of effectiveness.

1. Introduction

Mutation testing (DeMillo et al., 1978) is a testing technique that measures the quality of test cases and helps in designing new test cases. Mutation testing involves modifying a program by introducing simple syntactic changes and creating possible faulty versions, called *mutants*. Test cases are executed against both the original program and mutants. A mutant is *killed* by a test case that causes the mutant program to produce a different output from the original program's output. Test cases are considered to be effective if they kill most mutants.

Although mutation testing is powerful (Mathur and Wong, 1994; Frankl et al., 1997), its high execution cost has always been a problem to be solved. Several approaches have been proposed to reduce the cost of mutation testing. Cost reduction approaches are well reviewed in the survey papers (Jia and Harman, 2011; Usaola and Mateo, 2010). Among the approaches, reducing the number of mutants to be executed is the most obvious way to reduce costs. However, most approaches that run fewer mutants have the drawback that they are not as effective as mutation testing that uses a full set of mutants.

To address this issue, we propose a new approach that executes fewer mutants but is nearly as effective as approaches that use a full set of mutants. Our approach dynamically clusters mutants that are expected to produce the same results against a test case. For that, the execution of mutants whose mutated code is in a common position is halted immediately after executing the mutated code. The intermediate results are then compared, and mutants with identical intermediate results are clustered. A single mutant from each cluster is then fully executed using a strong mutation method against the test case. If the fully executed mutant is killed (or live), all remaining mutants in the cluster are considered to be killed (or live). The advantage of our approach is that it reduces the cost of mutation testing by restricting the number of mutants that are fully executed. In addition, it can lead to a further cost reduction by easily combining existing cost reduction approaches.

Our approach was implemented by extending a Java mutation system and experiments were conducted to determine the efficiency.

This paper is organized as follows. Sections 2 describes related work and Section 3 provides some background information in support of the new approach. Section 4, the main part of the paper, describes the cost reduction method for mutation testing. Section 5 presents experimental results and a cost comparison. Section 6 presents conclusions and discusses future work.

2. Related work

The number of mutants significantly affects the execution cost of mutation testing because each mutant is executed repeatedly against at least one (potentially many) test case. Many researches have attempted to reduce the number of mutants without significant loss of test effectiveness.

* Corresponding author. Tel.: +82 42 860 6551.
E-mail address: ysma@etri.re.kr (Y.-S. Ma).

http://dx.doi.org/10.1016/j.jss.2016.01.007
0164-1212/

A mutation survey paper (Jia and Harman, 2011) classified the approaches that use fewer mutants into four categories: (1) mutant sampling, (2) selective mutation, (3) higher order mutation, and (4) mutant clustering.

Mutant sampling (Acree, 1980; Budd, 1980) uses a small percentage, say *x%*, of randomly selected mutants and ignores the remaining mutants. An empirical study conducted by Wong (1993) showed that test sets adequate for 10% of randomly chosen mutants were only 16% less effective than mutation analysis that used a full set of mutants.

Selective mutation (Wong et al., 1994; Offutt et al., 1996) uses *selective mutation operators*, comprising portions of the entire mutation operators that are nearly as effective as non-selective mutation. Several studies (Wong et al., 1994; Offutt et al., 1996) have been conducted to identify efficient selective mutation operators. One of the widely used selective mutation operator set consists of five mutation operators (Offutt et al., 1996): ABS, UOI, LCR, AOR and ROR. Experimental trials showed that these five operators provide nearly the same coverage as non-selective mutation operators, with cost reduction of at least four times with small programs.

Higher order mutation (Jia and Harman, 2009; Polo et al., 2008; Mateo et al., 2013) uses higher-order mutants instead of first order mutants. The approach was originally proposed to identify higher-order mutants that denote subtle faults, and it also suggested subsuming higher order mutant which may be preferable to replace first order mutant.

Mutant clustering (Hussain, 2008; Ji et al., 2009) is an approach that selects a subset of mutants using a clustering algorithm. Each mutant in a cluster is likely to be killed by the same set of test cases; thus, one or a part of mutants from each cluster is used for mutation testing. The possibility of mutant clustering was first shown in the master's thesis of Hussain (2008), which clusters mutants by analyzing similarities among mutants. However, the method determines the similarity by executing all mutants repeatedly against all test cases. Ji et al. (2009) uses a domain analysis to determine similarity among mutants. The method uses symbolic execution to analyze the domain of variables; thus, its effectiveness is subordinate to the symbolic execution's ability.

This section introduces two additional categories for reducing the number of mutants: 'mutation subsumption' and 'dynamic mutant filtering' approaches.

Mutation subsumption (Kaminski et al., 2013; Just et al., 2012; Ammann et al., 2014; Kurtz et al., 2014) approach aims at not creating redundant mutants that are subsumed by other mutants. Kaminski et al. (2013) proposed that tests detecting three of the ROR (Relational Operator Replacement) mutants subsume (are guaranteed to detect) all seven ROR mutants; thus, the three ROR mutants were non-redundant. Just et al. (2012) investigated the subsumption relations among COR (Conditional Operator Replacement) mutants and suggested that only three COR mutants were non-redundant. Ammann et al. (2014) proposed a dynamic subsumption approach. They proposed a model for minimizing mutants with respect to a test set, thus, a minimal set of non-redundant mutants can be changed according to the used test set. Kurtz et al. (2014) define true subsumption, dynamic subsumption, and static subsumption to model the redundancy between mutants and develop a graph model to display the subsumption relationship.

Dynamic mutant filtering (Schuler and Zeller, 2009; Weiss and Fleyshgakker, 1993; Kim et al., 2013) is an approach which dynamically filters out mutants to be expected to be strongly live for each test case and improves the speed by excluding their execution. Schuler and Zeller (2009) executed only the reachable mutants with a code coverage analysis. Weiss and Fleyshgakker (1993) proposed an approach in which weak and strong mutations were combined for an interpretive mutation system. It filtered out weakly killed mutants. Kim et al.'s study (Kim et al., 2013) extended their approach to a non-interpretive mutation system. Experimental results showed that removing the execution of unreachable and weakly live mutants significantly reduced the cost of mutation testing.

The approach proposed in this paper takes advantage of both the dynamic mutant filtering approach and the mutant clustering approach. For each test case, weakly killed mutants are filtered, and weakly killed mutants with identical intermediate results are clustered. A single mutant from each cluster is then fully executed with the test case. The goal is to reduce the number of mutants that must be fully executed without reducing the test effectiveness.

3. Definition of conditionally overlapped mutants

We define the mutant M1 is *overlapped* to the mutant M2, and vice versa, if the mutants M1 and M2 are functionally identical. That is, the mutant M1 is an *overlapped mutant* of the mutant M2, and vice versa. Although the definition of an overlapped mutant is similar to that of an equivalent mutant, the concept is slightly different. An equivalent mutant (Yao et al., 2014) is a mutant that is functionally identical to the original program, thus it cannot be killed by any test case. On the other hand, an overlapped mutant is a mutant that is functionally identical to at least one other mutant and can be killed by some test cases if it is not an equivalent mutant. If a mutant is killed (or live), all of its overlapped mutants are killed (or live) in the same manner. Therefore, without executing all mutants, we can predict their results by running only one mutant from each set of overlapped mutants.

Consider the statement '$C = A + B$;' and its mutated versions of mutants $m1$ and $m2$: '$C = A - B$;' and '$C = A + (-B)$;'. In this example, the mutants $m1$ and $m2$ are overlapped. Executing both mutants would be a duplicated effort. The identification of overlapped mutants prior to execution would be beneficial; however, the complete detection of overlapped mutants is impossible because it is essentially the same problem as detecting equivalent mutants. Instead, we focus on a specific type of overlapped mutants, the *conditionally overlapped mutants*, described below.

Conditionally overlapped mutants are defined using a looser definition of the overlapped mutants. The mutant M1 is *conditionally overlapped (c-overlapped)* to the mutant M2 for a test case if the mutants M1 and M2 produce identical results for the test case. To avoid confusion, we will refer to overlapped mutants as *absolutely overlapped (a-overlapped) mutants*. A-overlapped mutants always produce identical results for any test cases; however, c-overlapped mutants are clustered depending on the test case, thus, different sets of c-overlapped mutants will be formed if a different test case is used.

Let us examine the program code in Fig. 1. Applying the ROR and AOR operators to the program yields 15 different mutated codes, as listed in Table 1. The ROR mutation operator is applied to the third line of code and produces seven mutants. The AOR mutation operator is applied to the fourth and sixth lines of code and produces eight mutants.

```
1  int myFunction(int A, int B) {
2     int C;
3     if ( A != B )
4        C = A + B;
5     else
6        C = A * B;
7     return C;
8  }
```

Fig. 1. An example code for conditionally overlapped mutants.