Information and Software Technology 000 (2016) 1–17

Contents lists available at ScienceDirect

Information and Software Technology

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

A path-aware approach to mutant reduction in mutation testing

Chang-ai Sun[a,*], Feifei Xue[a], Huai Liu[b], Xiangyu Zhang[c]

[a] School of Computer and Communication Engineering, University of Science and Technology Beijing, China
[b] Australia-India Research Centre for Automation Software Engineering, RMIT University, Melbourne, Australia
[c] Department of Computer Science, Purdue University, West Lafayette, IN, USA

ARTICLE INFO

Article history:
Received 1 August 2015
Revised 15 February 2016
Accepted 22 February 2016
Available online xxx

Keywords:
Mutation testing
Selective mutation testing
Control flow
Path depth

ABSTRACT

Context: Mutation testing, which systematically generates a set of mutants by seeding various faults into the base program under test, is a popular technique for evaluating the effectiveness of a testing method. However, it normally requires the execution of a large amount of mutants and thus incurs a high cost.

Objective: A common way to decrease the cost of mutation testing is mutant reduction, which selects a subset of representative mutants. In this paper, we propose a new mutant reduction approach from the perspective of program structure.

Method: Our approach attempts to explore path information of the program under test, and select mutants that are as diverse as possible with respect to the paths they cover. We define two path-aware heuristic rules, namely module-depth and loop-depth rules, and combine them with statement- and operator-based mutation selection to develop four mutant reduction strategies.

Results: We evaluated the cost-effectiveness of our mutant reduction strategies against random mutant selection on 11 real-life C programs with varying sizes and sampling ratios. Our empirical studies show that two of our mutant reduction strategies, which primarily rely on the path-aware heuristic rules, are more effective and systematic than pure random mutant selection strategy in terms of selecting more representative mutants. In addition, among all four strategies, the one giving loop-depth the highest priority has the highest effectiveness.

Conclusion: In general, our path-aware approach can reduce the number of mutants without jeopardizing its effectiveness, and thus significantly enhance the overall cost-effectiveness of mutation testing. Our approach is particularly useful for the mutation testing on large-scale complex programs that normally involve a huge amount of mutants with diverse fault characteristics.

1. Introduction

Mutation testing, basically a fault-based software testing technique [1,2], was originally proposed to measure the adequacy of a given test suite and help design new test cases to improve the quality of the test suite. It has been used for different purposes, such as the generation of test cases and oracles [3], fault localization [4], etc. Fig. 1 shows the principle of mutation testing. Given a base program, different variants, namely mutants, can be generated by seeding various faults through mutation operators. Once a test case shows different behaviors between a mutant and the base program, the mutant is said to be killed by the test case (in other words, the related fault is detected). Apparently, a test suite is regarded as effective if it can kill as many mutants as possible (i.e. large mutation scores). A number of studies [5–7] have shown that compared with manually fault-seeded programs, automatically generated mutants are more similar to the real-life faulty program. Thus, mutation testing has been acknowledged as an effective technique for evaluating the fault-detection capability of a testing method.

However, the real-world application of mutation testing is hindered by some drawbacks, such as the existence of equivalent mutants, lack of appropriate automated tools, etc. One major drawback is the high cost: Due to the large number of mutation operators and possible locations to apply these operators into the program, a huge volume of mutants are generated to guarantee that various faults are covered as many as possible. The execution of all these mutants is quite time-consuming, and the test result verification on mutants is a non-trivial task.

* Corresponding author. Tel.: +861062332931; fax: +861062332873.
E-mail addresses: casun@ustb.edu.cn, changai_sun2002@hotmail.com (C.-a. Sun), 729045626@qq.com (F. Xue), huai.liu@rmit.edu.au (H. Liu), xyzhang@cs.purdue.edu (X. Zhang).

http://dx.doi.org/10.1016/j.infsof.2016.02.006
0950-5849/

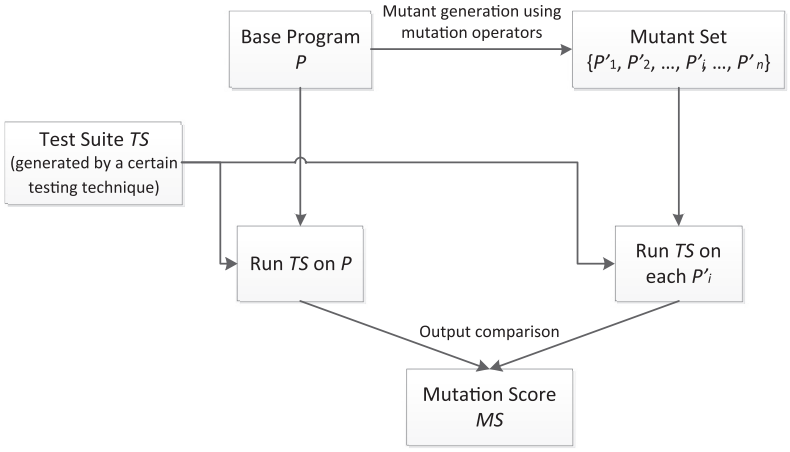

Fig. 1. Principle of mutation testing.

Some efforts have been made to decrease the cost of mutation testing by reducing the number of mutants. Mathur and Wong [8] proposed random mutant selection, which is simple and efficient in execution. However, random selection may discard some mutants that are difficult to be killed, and thus affect the quality of test suite that is designed based on the selected mutants. A more systematic approach called operator-based mutant selection [9] was proposed to select a subset of mutants based on certain (not all) mutation operators. Nevertheless, some recent studies [10] showed that the operator-based strategy is actually not superior to random selection.

In this paper, we propose a new mutant reduction approach. Instead of mutation operators, we conjecture that the fault characteristics (in particular, how different a fault is to be detected) are more related to the location of the fault, especially how deep the fault location is in terms of program paths. Therefore, we explore the mutant reduction based on the program structure. In particular, our work makes the following four contributions:

(I) A path-aware approach to mutant reduction is proposed, which explores mutant reduction from the perspective of the path depth in the program under test;
(II) We present four heuristic rules for mutant reduction, two of which are path-aware, one statement-based, and one operator-based;
(III) Four mutant reduction strategies are developed with different priorities among the heuristic rules; and
(IV) The effectiveness of the mutant reduction strategies are evaluated through an empirical study based on 11 real-life programs. It is shown that two strategies giving higher priorities to path-aware rules are superior to random mutant selection, and are more effective than the other two giving higher priorities to statement- or operator-based rules.

The rest of this paper is organized as follows. In Section 2, we introduce the underlying concepts and techniques. In Section 3, we describe our heuristic rules and the mutant reduction strategies. We present the design and setting of our empirical study in Section 4, and discuss the experimental results in Section 5. The work related to our study is discussed in Section 6. Finally, we conclude the paper in Section 7.

2. Preliminaries and terminology

In this section, we introduce the basic concepts and preliminaries that will be used by path-aware mutant reduction technique. All concepts are illustrated using an example program implementing heap sort, as shown in Fig. 2. The function call diagram of the program is given in Fig. 3.

Practically, a program is often composed of a number of modules, such as functions in C programs. We distinguish these modules into *caller* and *callee* based on the invoking relationship among them [11].

Definition 1. If module m directly invokes module n, module m is termed as the caller and module n the callee. The invoking relation is represented as $m \rightarrow n$.

Definition 2. $Callers(m)$ refers to the set of direct callers of module m, that is, $Callers(m) = \{x | x \rightarrow m\}$.

For example, in the heap sort program (Figs. 2 and 3), $f_1 \rightarrow f_2$, that is, between modules f_1 and f_2, f_1 is the caller, while f_2 is the callee. According to Fig. 3, we can get the following:

- $Callers(f_1) = \emptyset$.
- $Callers(f_2) = \{f_1\}$.
- $Callers(f_3) = \{f_1\}$.
- $Callers(f_4) = \{f_2\}$.
- $Callers(f_5) = \{f_2, f_4\}$.

We now define the module depth $MD(m_i)$ of a module m_i based on the invoking relation among modules.

Definition 3. For a module m_i,

$$MD(m_i) = \begin{cases} 0; & Callers(m_i) = \emptyset \\ \max\left(MD(m_j | m_j \in Callers(m_i))\right) + 1; & Callers(m_i) \neq \emptyset \end{cases}$$

For the heap sort program, we have the following calculations:

- Since $Callers(f_1) = \emptyset$, $MD(f_1) = 0$.
- Since $Callers(f_2) = \{f_1\}$, $MD(f_2) = \max(MD(f_1)) + 1 = 0 + 1 = 1$.
- Since $Callers(f_3) = \{f_1\}$, $MD(f_3) = \max(MD(f_1)) + 1 = 0 + 1 = 1$.
- Since $Callers(f_4) = \{f_2\}$, $MD(f_4) = \max(MD(f_2)) + 1 = 1 + 1 = 2$.
- Since $Callers(f_5) = \{f_2, f_4\}$, $MD(f_5) = \max(MD(f_2), MD(f_4)) + 1 = \max(1, 2) + 1 = 2 + 1 = 3$.

Note that there may exist recursive calls among modules, which in turn results in a cycle in the function call diagram. In this situation, we can break the cycle from the back edges in recursive calls, as discussed in [11].

Please cite this article as: C.-a. Sun et al., A path-aware approach to mutant reduction in mutation testing, Information and Software Technology (2016), http://dx.doi.org/10.1016/j.infsof.2016.02.006