Information and Software Technology 55 (2013) 866–879

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

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

Metamorphic slice: An application in spectrum-based fault localization ☆

Xiaoyuan Xie [a,c,d,*], W. Eric Wong [b], Tsong Yueh Chen [a], Baowen Xu [d]

[a] Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn, VIC 3122, Australia
[b] Department of Computer Science, University of Texas at Dallas, Richardson, TX 75083, United States
[c] School of Computer Science and Engineering, Southeast University, Nanjing 210096, China
[d] State Key Laboratory for Novel Software Technology & Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China

ARTICLE INFO

Article history:
Available online 28 August 2012

Keywords:
Spectrum-based fault localization
Slice
Metamorphic slice
Test oracle
Metamorphic testing

ABSTRACT

Context: Because of its simplicity and effectiveness, Spectrum-Based Fault Localization (SBFL) has been one of the popular approaches towards fault localization. It utilizes the execution result of *failure* or *pass*, and the corresponding coverage information (such as program slice) to estimate the risk of being faulty for each program entity (such as statement). However, all existing SBFL techniques assume the existence of a test oracle to determine the execution result of a test case. But, it is common that test oracles do not exist, and hence the applicability of SBFL has been severely restricted.
Objective: We aim at developing a framework that can extend the application of SBFL to the common situations where test oracles do not exist.
Method: Our approach uses a new concept of *metamorphic slice* resulting from the integration of metamorphic testing and program slicing. In SBFL, instead of using the program slice and the result of *failure* or *pass* for an individual test case, a metamorphic slice and the result of *violation* or *non-violation* of a metamorphic relation are used. Since we need not know the execution result for an individual test case, the existence of a test oracle is no longer a requirement to apply SBFL.
Results: An experimental study involving nine programs and three risk evaluation formulas was conducted. The results show that our proposed solution delivers a performance comparable to the performance observed by existing SBFL techniques for the situations where test oracles exist.
Conclusion: With respect to the problem that SBFL is only applicable to programs with test oracles, we propose an innovative solution. Our solution is not only intuitively appealing and conceptually feasible, but also practically effective. Consequently, test oracles are no longer mandatory for SBFL, and hence the applicability of SBFL is significantly extended.

1. Introduction

It is commonly recognized that testing and debugging are important but resource consuming activities in software engineering. Attempts to reduce the number of delivered faults in software are estimated to consume 50–80% of the total development and maintenance effort [2]. Locating faults is a tedious task, due to a great amount of manual involvement. This makes fault localization a major resource consuming task in the software development life cycle. Therefore many researchers have studied automatic and effective techniques for fault localization, in order to decrease its cost and to increase software reliability.

One promising approach for fault localization is Spectrum-Based Fault Localization (abbreviated as SBFL). Generally speaking, this approach utilizes various program spectra acquired dynamically from software testing, and the associated test result, in terms of *failed* or *passed*, for individual test cases. The program spectrum can be any granularity of program entities. One of the most widely adopted spectra is the execution slice (denoted as *e_slice*) of a test case, which records those statements executed in one test execution [3]. After collecting these data, SBFL uses different statistical formulas to evaluate the risk of containing a fault for each program entity, and gives a risk ranking list. SBFL intends to highlight program entities which strongly correlate with program failures. These program entities are regarded as the likely faulty locations [4]. Typical statistical formulas include Pinpoint, Tarantula, Ochiai, etc. [5–12].

SBFL has received a lot of attention due to its simplicity and effectiveness. However there are still some problems in this approach,

☆ A preliminary version of this paper was presented at the 11th International Conference on Quality Software (QSIC 2011) [1].

* Corresponding author at: Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn, VIC 3122, Australia. Tel.: +61 3 9214 8661; fax: +61 3 9819 0823.
E-mail addresses: xxie@swin.edu.au (X. Xie), ewong@utdallas.edu (W.E. Wong), tychen@groupwise.swin.edu.au (T.Y. Chen), bwxu@nju.edu.cn (B. Xu).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.08.008

and the assumption of the existence of a test oracle is one of the most crucial problems. In real world application domains, many programs have oracle problem, which include complex computational programs, machine learning algorithms, etc. [13,14]. In such programs, test results of individual test cases are not available and hence risk evaluation cannot be conducted. As a consequence, SBFL becomes inapplicable in these domains.

In this paper, we focus on the following research questions.

1. ***RQ1:*** How can we apply SBFL techniques to the software application domains without test oracle?
2. ***RQ2:*** How effective is our proposed solution?

For ***RQ1***, we propose a new type of slice, namely, metamorphic slice (*mslice*). Each *mslice* is related to a particular property of the algorithm being implemented. This property is referred to as the metamorphic relation (MR) which involves multiple inputs and their outputs. The notion of MR is introduced in the technique of metamorphic testing, which is proposed to alleviate the oracle problem. If the implementation of the algorithm is correct, MR must be satisfied with respect to the relevant multiple inputs and their outputs. For example, in *sin* function, property of $sin(x) = sin(x + 2\pi)$ can be used as an MR. If the outputs for any inputs x and $x + 2\pi$, are not equal, this MR is said to be violated and the implementation of *sin* is known to be incorrect. Therefore, for a particular MR, given a particular group of test cases, a test result about whether the MR is *violated* or *non-violated*, can be determined, even when we do not know whether the test result is *failed* or *passed* for each element of this group of test cases. In other words, the role of *failed* or *passed* for an individual test case can be replaced by the role of *violation* or *non-violation* for a group of test cases in SBFL. As a consequence, SBFL can still be applied in the absence of a test oracle.

Conceptually speaking, our proposed approach can obviously extend the application of existing SBFL techniques to programs without test oracles. But it is still important to know whether it is feasible in practice. Thus, for ***RQ2***, we conducted experimental studies to investigate the performance of our approach. For programs with test oracle, the performance of existing SBFL techniques has been regarded to be satisfactory by the community. Therefore, it is natural to compare the performance of our approach with the performance of the existing SBFL techniques. In essence, we are interested in seeing whether there is a significant impact when the conventional program slice is replaced by *mslice* and the test result of *failed* or *passed* is replaced by the test result of *violation* or *non-violation*.

The rest of this paper is organized as follows: Section 2 introduces the background of SBFL and metamorphic testing. Section 3 gives the concept of *metamorphic slice*, and also describes how it is applied in SBFL to alleviate the oracle problem. In Section 4, we present the experimental set-up for our case study. We analyze the empirical results and threats to validity in Section 5 and Section 6, respectively. Section 7 describes the conclusions and future work.

2. Background

2.1. Spectrum-based fault localization (SBFL)

In SBFL, two types of information are essential, namely, program spectrum and test results. A program spectrum is a collection of data that provides a specific view on the dynamic behavior of software [5]. Generally speaking, it records the run-time profiles about various program entities for a specific test suite. The program entities could be statements, branches, paths, basic blocks, etc.; while the run-time information could be the binary coverage status, the number of times that the entity has been covered, the program state before and after executing the program entity, etc. The other essential information is the test result of *failed* or *passed* associated with each test case. Together with the coverage information, the test results give debuggers hints about which program entities are more likely to be related to failure, and hence to contain faults. Obviously, there are many kinds of combinations of these two types of data [15]. The most widely adopted combination involves statement and its coverage status in one test execution [3], which will also be used in this study.

Given a program P with n statements and m test cases, Fig. 1 shows the essential information required by SBFL. In Fig. 1, vector TS contains the m test cases, matrix MS represents the program spectrum and RE records all the test results associated with individual test cases. The element in the ith row and jth column of matrix MS represents the coverage information of statement s_i, by the test case t_j, with 1 indicating s_i is executed, and 0 otherwise. And the jth element in RE (denoted as r_j) represents the test result of t_j in TS, with p indicating *passed* and f meaning *failed*. For each statement s_i, the relevant program spectrum and test results are represented as a vector of four elements, denoted as $A_i = \langle a^i_{ef}, a^i_{ep}, a^i_{nf}, a^i_{np} \rangle$, where a^i_{ef} and a^i_{ep} represent the number of test cases in TS that execute statement s_i and return the test results of *fail* or *pass*, respectively; a^i_{nf} and a^i_{np} denote the number of test cases that do not execute s_i, and return the test results of *fail* or *pass*, respectively. Obviously, the sum of these four parameters for each statement should always be equal to m.

A risk evaluation formula R is then applied on each statement s_i to map A_i into a real value that indicates the risk of being faulty for s_i. Existing evaluation formulas include Jaccard [7], Ochiai [10], Ample [12], Tarantula [6], Wong [11], etc. A statement with higher risk value is expected to be more likely to be faulty, which therefore should be examined with higher priority. Hence, after assigning the risk values to all statements, the statements are sorted descendingly according to their risk values. Debuggers are supposed to inspect the statements according to this ranking list from top to bottom.

An example is shown in Fig. 2, in which, program PG has four statements $\{s_1, s_2, s_3, s_4\}$, and test suite TS has six test cases $\{t_1, t_2, t_3, t_4, t_5, t_6\}$. t_5 and t_6 give rise to *failed* runs and the remaining four test cases give rise to *passed* runs, as indicated in RE. Matrix MS records the binary coverage information for each statement with respect to every test case. Matrix MA is defined so that its ith row represents the corresponding A_i for s_i. For instance, in this figure, $a^1_{np} = 0$ means that no test case in the current test suite gives a test result of *pass* without executing s_1; $a^4_{ef} = 2$ represents that s_4 is executed by two test cases which can detect failure.

Consider a widely investigated formula Tarantula as an illustration, whose expression is shown as follows [6]:

$$R_T(s_i) = \frac{a^i_{ef}}{a^i_{ef} + a^i_{nf}} \Big/ \left(\frac{a^i_{ef}}{a^i_{ef} + a^i_{nf}} + \frac{a^i_{ep}}{a^i_{ep} + a^i_{np}} \right)$$

$$TS: (t_1 \quad t_2 \quad \dots \quad t_m)$$

$$PG: \begin{pmatrix} s_1 \\ s_2 \\ \cdot \\ \cdot \\ \cdot \\ s_n \end{pmatrix} \quad MS: \begin{pmatrix} 1/0 & 1/0 & \dots & 1/0 \\ 1/0 & 1/0 & \dots & 1/0 \\ & \cdot & & \\ & \cdot & & \\ & \cdot & & \\ 1/0 & 1/0 & \dots & 1/0 \end{pmatrix}$$

$$RE: (p/f \quad p/f \quad \dots \quad p/f)$$

Fig. 1. Essential information for SBFL.

$$TS:(t_1\ t_2\ t_3\ t_4\ t_5\ t_6) \qquad A_i:(a^i_{ef}\ a^i_{ep}\ a^i_{nf}\ a^i_{np})$$

$$PG:\begin{pmatrix}s_1\\ s_2\\ s_3\\ s_4\end{pmatrix}\quad MS:\begin{pmatrix}1&1&1&1&1&1\\ 0&0&0&1&0&0\\ 0&1&1&1&0&1\\ 1&1&0&1&1&1\end{pmatrix}\quad MA:\begin{pmatrix}2&4&0&0\\ 0&1&2&3\\ 1&3&1&1\\ 2&3&0&1\end{pmatrix}$$

$$RE:(p\ \ p\ \ p\ \ p\ \ f\ \ f)$$

Fig. 2. An example.

Given the matrix *MS* in Fig. 2, Tarantula assigns risks values for s_1, s_2, s_3 and s_4 as $\frac{1}{2}$, 0, $\frac{2}{5}$ and $\frac{4}{7}$, respectively. After ranking these statements descendingly according to their risk values, we have a list of "s_4, s_1, s_3, s_2". Debuggers are expected to examine statements from the top of this list.

SBFL is widely adopted because it is simple in concept and easily applied. Furthermore, experimental analysis shows that it is effective. However, in order to evaluate the risk values, SBFL requires information of program spectrum that is associated with conventional test results of *fail* or *pass*. In other words, all the existing SBFL techniques have assumed the existence of a test oracle. Actually, such an assumption is not always true. In real-world applications, many programs have the oracle problem, that is, it is impossible or too expensive to verify the correctness of the computed outputs. For example, when testing a compiler, it is not easy to verify whether the generated object code is equivalent to the source code. Other examples include testing programs involving machine learning algorithms, simulations, combinatorial calculations, graph display in the monitor, etc. [13,14]. In such situations, test result of each individual test case is unavailable, and hence there is insufficient information for risk evaluation. As a consequence, current SBFL techniques become inapplicable in these domains.

2.2. Metamorphic testing

Metamorphic testing (MT) [16] is a methodology designed to alleviate the oracle problem. Instead of verifying the correctness of the computed outputs of individual test cases, MT uses some specific properties of the problem domain, namely metamorphic relations (MRs), to verify the relationship between multiple but related test cases and their outputs.

Given a program P under test, some MRs are first defined. An MR is a property of the algorithm, involving multiple test case inputs and their outputs. Amongst these multiple inputs, some inputs have their values defined or selected by some test case selection strategies, and these inputs are referred to as source test cases. Once the source test cases are defined, the values of the remaining inputs involved in the MR can then be defined. Such inputs are known as follow-up test cases. The key point of MT is that though we may not know the correctness of the outputs of individual inputs, we are able to verify the validity of the MR after executing the program with all source test cases and follow-up test cases.

Let us use an example to illustrate MT informally. Readers who are interested in a more comprehensive description of MT, may consult [16]. Consider a program that searches for the shortest path between any two nodes in an undirected graph and reports its length. Let us denote the length of the shortest path as $d(x,y,G)$, where G is a weighted undirected graph, x is the start node, and y is the destination node. Suppose that $d(x,y,G)$ is computed as 13,579. It is very expensive to check whether 13,579 is correct due to the combinatorially large number of possible paths between x and y. Therefore, such a problem is said to have the oracle problem. When applying MT to this program, we first need to define an MR based on some well-known properties in graph theory. One possible MR (referred as MR1) is that $d(x,y,G) = d(y,x,G)$. Another possible MR (referred as MR2) is that $d(x,y,G) = d(x,w,G) + d(w,y,G)$, where w is any node in the shortest path with x as the start node and y as the destination node.

The core idea is that although it is difficult to verify the correctness of the individual output, namely $d(x,y,G)$, $d(y,x,G)$, $d(x,w,G)$ and $d(w,y,G)$, it is easy to verify whether the MR1 and MR2 are satisfied or not, that is, whether $d(x,y,G) = d(y,x,G)$ and $d(x,y,G) = d(x,w,G) + d(w,y,G)$. If $d(y,x,G)$ is not equal to 13,579, then the program must be incorrect. However, if $d(x,y,G)$ is also 13,579, we cannot determine whether the program is correct or not. But, this is the limitation of software testing. In this example, (w,y,G) is referred to as the source test case, (y,x,G) is the follow-up test case of MR1, and both (x,w,G) and (w,y,G) are the follow-up test cases of MR2. As shown, follow-up test cases could be multiple and are not only dependent on the source test case but also on the relevant MR. Similarly, source test cases could be multiple in an MR.

Generally speaking, when conducting MT, the testers first need to identify an MR of the software under test, and choose a test case selection strategy to generate source test cases. For convenience of reference, we will refer to a source test case (or a group of source test cases if appropriate) and its related follow-up test cases as a metamorphic test group. Then, the program is executed with all test cases of a metamorphic test group. However, the correctness of the output of each individual test case need not be verified. Instead, the MR is verified with respect to the metamorphic test group and its outputs. *Violation* of MR implies an incorrect program.

Obviously, metamorphic testing is simple in concept, easily automatable, and independent of programming languages. In MT, the most crucial step is the identification of the MRs. Admittedly, this is not a trivial task. However, recent studies have provided empirical evidence that after a brief general training, testers can properly define MRs and effectively apply MT on the target programs [17,18]. This observation is consistent with the original expectation for MT. After all, in any case, a tester must have domain knowledge on the program under test. Therefore, at least some basic properties of the program can be determined, which are normally sufficient to construct MRs. Actually, almost all of the MRs in previous studies are derived from the basic properties [19–21,13,14].

Moreover, there are methods that can facilitate the identification of MRs. For example, when testing a group of programs which serve similar purpose, an MR repository can be constructed by harnessing the domain knowledge that indicates some general anticipation in this domain. And MRs from such repository could be generally reused in this area [14].

Since MR plays a key role in MT, identification of MRs deserves further study. However, this is beyond the scope of this paper. Interested readers may consult [16,22,23,17,14,18] for some previous related works.

3. Approach

3.1. Metamorphic slice

Main types of slices include *static slice*, *dynamic slice*, *execution slice* and *conditioned slice* [24–26,3]. *Dynamic slice* and *execution slice* have been widely used in debugging and their definitions are as follows:

- Given a variable v and a test case t, a *dynamic slice* denoted as $d\_slice(v,t)$, is the set of statements that have actually affected the variable v in the test run with t.

- Given a test case t, an *execution slice* denoted as $e\_slice(t)$, is the set of statements that have been covered in the test run with t.

As mentioned above, when using either of these two slices in SBFL, we must know the test outcome of *failure* or *pass* associated with each test case t. However in reality, such information may not be available if the program under test has the oracle problem. Since metamorphic testing has been proposed to alleviate the oracle problem, it is natural to consider how to integrate the concepts of the slices with the concept of metamorphic testing, to support the application of SBFL to application domains having the oracle problem. Therefore, we propose a new concept of *metamorphic slice* (abbreviated as *mslice*) to achieve this goal.

Generally speaking, an *mslice* is a group of slices which are bound together with a specific MR. In our context of SBFL, it is the coverage information which affects the risk evaluation. Thus, it is intuitively appropriate to use the set union operation to group relevant slices in the context of SBFL. Hence, corresponding to *d_slice* and *e_slice*, we define *dynamic mslice* and *execution mslice* for SBFL as follows:

Definition 1. For a metamorphic relation MR, suppose $T^S = \{t_1^S, t_2^S, \ldots, t_{ks}^S\}$ and $T^F = \{t_1^F, t_2^F, \ldots, t_{kf}^F\}$ are its respective set of source test cases and set of follow-up test cases, such that T^S and T^F constitute a metamorphic test group g.

- Given a variable v, the *dynamic mslice*, $d\_mslice(v,MR,T^S)$ is the union of all $d\_slice(v,t)$, where t is a test case of the metamorphic test group.

$$d\_mslice(v,MR,T^S) = \left(\bigcup_{i=1}^{ks} d\_slice(v,t_i^S)\right) \cup \left(\bigcup_{i=1}^{kf} d\_slice(v,t_i^F)\right)$$

- The *execution mslice*, $e\_mslice(MR,T^S)$ is the union of all $e\_slice(t)$, where t is a test case of the metamorphic test group.

$$e\_mslice(MR,T^S) = \left(\bigcup_{i=1}^{ks} e\_slice(t_i^S)\right) \cup \left(\bigcup_{i=1}^{kf} e\_slice(t_i^F)\right)$$

The following example illustrates how to construct an *e_mslice* from *e_slices* of the source and follow-up test cases. Without loss of generality, let us consider an MR involving one source test case and one follow-up test case. Suppose the program under test has 10 statements $\{s_1, s_2, \ldots, s_{10}\}$ and the *e_slices* of the source and follow-up test cases are $\{s_1,s_2,s_3,s_5,s_{10}\}$ and $\{s_1,s_2,s_6,s_7,s_{10}\}$, respectively. Immediately from Definition 1, the *e_mslice* of this metamorphic group is the union of these two *e_slices*, that is, $\{s_1,s_2,s_3,s_5,s_6,s_7,s_{10}\}$.

Technically speaking, an *mslice* has bound the $d\_slice(v,t)$ (or $e\_slice(t)$) of all test cases belonging to a metamorphic test group of MR. More importantly, regardless of the availability of the test result associated with each *d_slice* or *e_slice*, a metamorphic test result of *violation* or *non-violation* is always available for each *d_mslice* or *e_mslice*. As a consequence, we are able to extend the application of SBFL beyond the programs that have test oracles.

3.2. SBFL with e_mslice

In SBFL with *e_mslice*, the coverage information is provided by the *e_mslice* of each metamorphic test group g_i. Suppose there are m metamorphic test groups in the current metamorphic test suite. Then, there are m *e_mslices* and m metamorphic test results in total. With this information, we can construct the relevant matrix and vectors as the conventional SBFL, in Fig. 3.

In Fig. 3, the vector *MTS* denotes the test suite containing m metamorphic test groups. The jth column of matrix *MS* represents

$$MTS:(g_1 \quad g_2 \quad \ldots \quad g_m)$$
$$PG:\begin{pmatrix} s_1 \\ s_2 \\ \cdot \\ \cdot \\ \cdot \\ s_n \end{pmatrix} \quad MS:\begin{pmatrix} 1/0 & 1/0 & \ldots & 1/0 \\ 1/0 & 1/0 & \ldots & 1/0 \\ & & \cdot & \\ & & \cdot & \\ & & \cdot & \\ 1/0 & 1/0 & \ldots & 1/0 \end{pmatrix}$$
$$RE:(v/n \quad v/n \quad \ldots \quad v/n)$$

Fig. 3. Essential information for SBFL with *e_mslice*.

the corresponding *e_mslice* of g_j, in which the binary value of "1" in the ith line denotes the membership of statement s_i in this *e_mslice*; and 0 otherwise. Besides, the jth element in vector *RE* records the corresponding metamorphic test result for g_i, with "v" indicating *violated* and "n" indicating *non-violated*.

The transformation from Fig. 1 (information for SBFL with *e_slice*) to Fig. 3 (information for SBFL with *e_mslice*) consists of the following replacements. Each test case t_j is replaced by a metamorphic test group g_j; the *e_slice* for t_j is replaced by *e_mslice* for g_j; and the test result of *failed* or *passed* is replaced by the metamorphic test result of *violated* or *non-violated*. After such replacements, the same procedure can be applied to reformulate the vector and then evaluate the risk value r_i for each statement s_i.

In the conventional SBFL using *e_slice*, a *failed* test case implies that a faulty statement is definitely included in the relevant *e_slice*; while a *passed* test case does not provide a definite conclusion whether the relevant *e_slice* is free of faulty statement. Similarly, a *violated* metamorphic test group implies that there is at least one *failed* test case within it. Even though we do not know which test cases are actually the failed ones, we still can conclude that a faulty statement must be included in the union of all the related *e_slices*, that is, the *e_mslice*. On the other hand, a *non-violated* metamorphic test group does not provide a definite conclusion about whether it involves any failed test case and as a consequence, the correctness of all statements in the current *e_mslice* is not guaranteed.

4. Experimental set-up

4.1. Test objectives and their metamorphic relations

In the investigation of the effectiveness of our approach, we have selected nine programs of different sizes as our test objectives. Table 1 lists the number of the executable lines of codes (eLOC) excluding those statements such as blank lines and lines of left or right brace. The data is collected by SLOCCount (version 2.26) from [27]. For each program, we have enumerated three MRs.

Table 1
Executable line of codes.

Program	eLOC
grep	7309
SeqMap	1783
print_tokens	342
print_tokens2	355
replace	512
schedule	292
schedule2	269
tcas	135
tot_info	273

4.1.1. grep

grep is a well-known UNIX utility written in C to perform pattern matching. Given a pattern to be matched and some input files for searching, this program searches these files for lines containing a match to the specified pattern. When a match in a line is found, the whole line is printed to standard output. For example, we can use: *grep "[Gg]r?ep" myfile.txt*, where expression "*[Gg]r?ep*" is the specified regular expression to be matched, and "*myfile.txt*" is the input file. *grep* searches "*myfile.txt*" for lines containing a match to pattern "*[Gg]r?ep*", and prints all lines containing "*grep*", "*Grep*", "*gep*" or "*Gep*".

However, testing *grep* is not a easy task, because it may be very difficult to verify the correctness of its output. As shown in the above example command line, though we can check whether all the printed lines actually contain matchings to the specified pattern, it is almost impossible to know whether *grep* has printed all the matched lines, unless we do an exhaustive examination of the entire file (namely, inspecting every single line of myfile.txt). Therefore, *grep* has the oracle problem, which makes it a suitable objective for our study.

The source code for *grep* used in our experiments is v0 in *grep1.2* that is obtained from the Software-artifact Infrastructure Repository (SIR) website [28].

For this program, our MRs are related to the regular expression analyzer, which is one of its key components. Given the same input file to be scanned, the regular expressions in the source and follow-up test cases are denoted as "r_s" and "r_f", respectively. All the three MRs involve r_f which is equivalent to r_s. As a consequence, the output of the follow-up test case (denoted as O_f) should be the same as the output of the source test case (denoted as O_s).

(1) **MR1: Completely decomposing the bracketed sub-expression**
In a bracketed sub-expression "$[x_1 \cdots x_n]$", where x_i is a single character, if these characters "$x_1, \ldots, x_n$" are continuous for the current locale, they can be presented in a condensed way "$[x_1\text{–}x_n]$". In our experiments, we consider the default C locale, where characters are sorted according to their ASCII codes. For such a bracketed sub-expression, one of its equivalents is the complete decomposition of the bracket, by using the symbol "|" that means "or". In MR1, we construct r_f by completely decomposing such bracketed sub-expressions in r_s. For example, if r_s contains a sub-expression "[*abcdef*]" or "[*a–f*]", then we have "*a*|*b*|*c*|*d*|*e*|*f*" instead in r_f.

(2) **MR2: Splitting the bracketed structure**
Consider the bracketed sub-expression "$[x_1 \cdots x_n]$" or "$[x_1\text{–}x_n]$" again. Another equivalent format is to split the bracket into two brackets, by using symbol "|". In MR2, r_f is constructed by replacing such sub-expression in r_s with this equivalent. For example, if r_s contains a sub-expression "[*abcdef*]" or "[*a–f*]", then we have "[*ab*]|[*c–f*]" instead in r_f.

(3) **MR3: Bracketing simple characters**
Apart from the reserved words with special meanings, any simple character should be equivalent to itself enclosed by the brackets, that is, "*a*" is equivalent to "[*a*]" if *a* is not a reserved word. In MR3, r_f is constructed by replacing some simple characters in r_s with their bracketed formats. For example, if r_s contains a sub-expression "*abc*", then we have "[*a*][*b*][*c*]" instead in r_f.

4.1.2. SeqMap

Another program used in our experiment is *SeqMap*, a Short Sequence Mapping Tool in bioinformatics [29]. Given a long reference string t and a set of short strings $P = \{p_1, \ldots, p_k\}$, which are finite strings of characters taken from the set of alphabets $\{A,T,G,C\}$, as well as a maximum number of mismatches e, *SeqMap* finds all substrings in t such that each substring has an edit distance equal to or less than e against some $p_i \in P$. Here edit distance refers to the number of operations required to transform one string to another. The valid edit operations include substitution, insertion and deletion. If a p_i matches any substring in t with not more than e edit distance, it is said to be mappable, otherwise unmappable. *SeqMap* outputs all the mappable p_i with optional information including the mapped location in t, the mapped substring of t, the edit distance of this mapping, etc.

Obviously, for each mappable p_i, it may not be difficult to verify the correctness of the printed mapping information. However, it is very expensive to check whether *SeqMap* has printed all the possible matching positions in t, or whether all the unmappable p_i are indeed truly unmappable to t. In other words, soundness of the result is easy to verify, but not the completeness of the result. Therefore, *SeqMap* also has the oracle problem.

The program used in our experiments is version 1.0.8, which contains 10 head files and 1 source file (in C++). We only targeted the following three files: probe.h, probe_match.h, and match.cpp, which implement the major functionalities of the program.

For this program, we construct MRs by modifying t and e while keeping P unchanged. Given a set of short strings $P = \{p_1, \ldots, p_k\}$, a long reference string t_s and the specified maximum of mismatches e_s as a source test case, the output is denoted as M_s. Obviously, $M_s \subseteq P_s$. And the set of unmappable short strings is $U_s = (P \backslash M_s)$. Let us denote the long reference string and the specified maximum of mismatches in the follow-up test case as t_f and e_f, respectively. The sets of mappable and unmappable short strings produced by the follow-up test case are referred to as M_f and U_f, respectively.

(1) **MR1: Concatenation of some elements of P to t_s**
Suppose P_1 is any non-empty subset of P. t_f is constructed by concatenating all elements in P_1 to the end of t_s one by one. As a consequence,
- For any $p_i \in M_s$, we have $p_i \in M_f$. Thus, $M_s \subseteq M_f$.
- For each $p_i \in (M_s \cap P_1)$, the follow-up test case should have at least one additional mapping location in t_f.
- Each $p_i \in (U_s \cap P_1)$ should be mapped at least once in t_f, that is, for such p_i, we have $p_i \in M_f$.

(2) **MR2: Deletion of substrings in t_s**
In this MR, t_f is constructed from t_s by deleting an arbitrary portion of strings at either the beginning or the end of t_s. As a consequence, for any $p_i \in U_s$, we have $p_i \in U_f$. Therefore, $U_s \subseteq U_f$.

(3) **MR3: Changing of e_s**
In this MR, $t_f = t_s$. And e_f can be set to either greater or smaller than e_s.
- Consider the case that $0 \leqslant e_f < e_s$. Then, we have $M_f \subseteq M_s$.
- Consider the case that $0 \leqslant e_s < e_f$. Then, we have $M_s \subseteq M_f$.

4.1.3. Siemens Suite

Besides, we also selected 7 small-scaled programs from the *Siemens Suite*, which have been extensively used in previous SBFL studies.

4.1.3.1. Print_tokens and print_tokens2. These two programs perform lexical parsing. They both read a sequence of strings from a file, group these strings into tokens, identify token categories and print out all the tokens and their categories in order. The main difference between these two programs is that *print_tokens* uses a hard-coded DFA; while *print_tokens2* does not.

Suppose the input files in source test case and follow-up test case are denoted as I_s and I_f, respectively, and their respective outputs are denoted as O_s and O_f. Each element in O_s and O_f has two attributes: the token category (e.g. keyword, identifier, etc.) and

the string of this token. For these two programs, we define MRs as follows.

(a) **MR1: Changing lower case into upper case**
In this MR, I_f is constructed from I_s by changing all characters in I_s with lower cases into their upper cases. Since *print_tokens* attempts to identify tokens and their categories, we have the size of O_f equal to the size of O_s. Besides, since all "keywords" are case-sensitive, all the elements with categories of "keyword" in O_s become "identifier" in O_f. For the non-keyword elements of O_s, the corresponding categories remain the same in O_f.

(b) **MR2: Deletion of comments**
In this MR, I_f is constructed from I_s by deleting all comments in I_s. Then, we have $O_s = O_f$.

(c) **MR3: Insertion of comments**
In this MR, I_f is constructed from I_s by inserting the comment symbol ";" at the very beginning of some arbitrarily chosen lines. Then, we have $O_f \subseteq O_s$.

4.1.3.2. Replace. Program *replace* performs regular expression matching and substitutions. It takes a regular expression *r*, a replacement string *s* and an input file as input parameters. It produces an output file resulting from replacing any substring in the input file that matched by *r*, with *s*. Instead of adopting the widely used Perl regular expression syntax, *replace* has its own syntax of regular expression.

Similar to *grep*, for *replace*, our MRs are also related to the regular expression analyzer. Given the same replacement string and input file, the regular expressions in the source and follow-up test cases are denoted as "r_s" and "r_f", respectively. All three MRs involve r_s and r_f that are equivalent. As a consequence, the output of the follow-up test case should be the same as the output of the source test case. However, since the syntax for regular expression in *replace* does not completely comply with the Perl regular expression syntax, not every MR for *grep* can be used for *replace*.

(a) **MR1: Extension and permutation in brackets**
The syntax of regular expression in *replace* also supports the bracketed sub-expression "$[x_1 \cdots x_n]$", where x_i is a single character. Such an expression has an equivalent format "$[x_1\text{–}x_n]$", if "$x_1, \ldots, x_n$" are continuous for the current locale. Thus, this MR constructs r_f with two types of replacements.
- The condensed sub-expression "$[x_1\text{–}x_n]$" in r_s is replaced by its extended equivalent "$[x_1 \cdots x_n]$".
- The extended sub-expression "$[x_1 \cdots x_n]$" in r_s is replaced by "$[x_{i_1} \cdots x_{i_n}]$", where "$\langle x_{i_1}, \ldots, x_{i_n} \rangle$" is a permutation of "$\langle x_1, \ldots, x_n \rangle$".

For example, if r_s contains a sub-expression "[*a*–*c*]", then we have "[*abc*]" instead in r_f. For another example, if r_s contains a sub-expression "[*abc*]", then r_f has "[*cba*]" instead.

(b) **MR2: Bracketing simple characters**
This MR is basically the same as the MR3 of *grep*. As a reminder, *grep* and *replace* have different lists of reserved words.

(c) **MR3: Replacement of escape character with bracketed structure**
In *replace*, symbol "@" is the escape character. Any reserved character, like "$" or "%" after "@" stands for its original meaning. Actually there is another way to preserve the original meaning for these reserved words, by bracketing the reserved characters individually. For example "@$" and "[$]" both mean "$". In MR3, the r_f is constructed by replacing the escaped reserved words with "@" in r_s with the bracketed format.

4.1.3.3. Schedule and schedule2. These two programs perform priority scheduling, which internally maintain four mutually exclusive job lists:

- Three priority job-lists P_1, P_2 and P_3, with P_3 and P_1 indicating the highest and lowest priorities, respectively: each list contains a list of jobs with the same priority.
- One blocked job list P_B: it contains all jobs currently suspended.

There is one and only job pointer that points to the currently activated job p_a, which is the first job in P_i with the highest priority in the system. At any time there is at most one activated job in the system. Each job has two attributes: the Job_ID and the Job_Priority. The Job_ID is unique for a job. For a newly created job, the Job_ID is assigned in an incremental manner. The Job_Priority indicates the priority value of 1, 2 or 3.

The basic functionality of *schedule* and *schedule2* is the same, except that *schedule* is non-preemptive and *schedule2* is preemptive. Both of them accept three integers (a_1, a_2 and a_3) and an input file as the input parameters, where a_i specifies the initial number of jobs in P_i. The input file contains a series of commands that specify various operations on the job lists. There are seven types of operations including addition of job, deletion of job, increasing the priority of a job, etc. The operations are coded from 1 to 7.

Given the same a_i ($1 \leqslant i \leqslant 3$), the input files containing a series of commands in the source test case and follow-up test case are denoted as C_s and C_f, respectively. For *schedule*, we define the MRs as follows.

(a) **MR1: Substitution of the quantum expire command**
In *schedule*, command coded in integer "5" is called "QUANTUM_EXPIRE" command, which releases the currently activated job and puts it to the end of the corresponding job list. Command coded in integer "3" is called "BLOCK" command, which puts the currently activated job to the block list P_B. Its reverse command "UNBLOCK" command is coded in "4" that takes a ratio *r* as a parameter. The index of the selected job to be unblocked from P_B is determined by multiplying the length of P_B by the ratio *r*. When processing command "4 *r*", *schedule* first releases the selected job from P_B, and then put it to the end of the corresponding job list. Therefore, command "5" can be re-interpreted as command "3", followed by "4 1.00". By using "3" and "4 1.00", *schedule* first blocks the currently activated job (denoted as p_a), puts it at the end of P_B, then unblocks the last job in P_B, i.e. p_a, (since the length of P_B multiplied by 1.00 indicates the last index in P_B) and puts this job to the end of the corresponding job list. Obviously, consecutively processing these two commands has the same effect as solely processing command "5". Thus, in MR1, C_f is constructed by replacing command "5" in C_s with commands "3" and "4 1.00". The output of the follow-up test case (denoted as O_f) should be the same as the output of the source test case (denoted as O_s).

(b) **MR2: Substitution of the adding job command**
In *schedule*, command coded in "1" is called "NEW_JOB" command, which takes an integer *i* ($1 \leqslant i \leqslant 3$) as its parameter to specify the priority, and adds a new job to the end of P_i. And command coded in "2" is the "UPGRADE_PRIO" command, which promotes a job from its current priority job list (P_i) into the next higher priority job list (P_{i+1}), where $i = 1$ or 2. This command takes two parameters. The first one is an integer *i* of 1 or 2, which specifies the current priority job list P_i. The second parameter is a ratio *r*, and the index of the selected job in P_i to be upgraded is determined by multiplying the length of P_i by the ratio *r*. As a consequence, directly adding a new job with priority $(i + 1)$ has the same effect as

adding a job with priority i and then promoting it to the job list with priority $(i+1)$. Thus, in MR2, C_f is constructed by replacing command "1 i + 1" in C_s with command "1 i" followed by "2 i 1.00" (i = 1 or 2). O_f should be the same as O_s.

(c) **MR3: Substitution of block and unblock commands**
As mentioned above, commands "3" and "4" are the reverse of each other. Command "3" increases the job number in P_B; while command "4" has the opposite effects. Therefore, we can make some replacements on these two types of commands, without changing the number of blocked jobs.
Suppose that a list of consecutive commands consists of commands "1", "2", "3", "4" or "5", and the numbers of commands "3" and "4" in this list are m and n, respectively. Besides each of these two types of commands has actually blocked or unblocked a job, rather than operating on a "NULL" job pointer.[1] Then, after processing this command list, there should be $k = m - n$ jobs in P_B. Therefore, we can remove all the commands "3" and "4" and insert the following commands in this command list, without changing the value of k:

- If $k > 0$, insert command "3" k times.
- If $k < 0$, insert command "4" k times (the parameter of ratio can be any value within [0.00, 1.00]).
- If $k = 0$, insert no command.

In MR3, we search for such sub-lists of commands in C_s and apply the above removal and insertion operations to construct C_f. The number of printed jobs in O_f and O_s must be the same.

And for *schedule2*, we also use the above MRs. However, since *schedule2* is a preemption scheduler, rescheduling happens in *schedule2* but not in *schedule*, whenever a job is added, unblocked or promoted to a higher priority list. Thus, in each MR of *schedule2*, only the number of printed jobs in O_f and O_s, but not necessarily their contents, must be the same.

4.1.3.4. Tcas. This program is a module of an on-board aircraft conflict detection and resolution system used by commercial aircraft. The entire system continuously monitors the radar information to check whether there is any neighbor aircraft (called intruder) that may give rise to a dash. *tcas* takes 12 integer parameters, including the altitudes of the controlled aircraft and the intruder aircraft, etc. It outputs a *Resolution Advisory* (*RA*) to advise the pilot to climb (*UPWARD*), descend (*DOWNWARD*) or remain the current trajectory (*UNRESOLVEDWARD*), as follows.

- if $A \wedge B \wedge C$ is true, then *RA* = *UPWARD*;
- if $A \wedge \neg B \wedge D$ is true, then *RA* = *DOWNWARD*;
- otherwise, *RA* = *UNRESOLVEDWARD*.

where

$$A = (High\_Confidence = 1) \wedge (Own\_Tracked\_Alt\_Rate \leqslant 600) \wedge (Cur\_Vertical\_Sep > 600) \wedge (A_a \vee A_b) \quad (1)$$

$$A_a = (Other\_Capability = 1) \wedge (Two\_Of\_Three\_Report\_Valid = 1) \wedge (Other\_RAC = 0) \quad (2)$$

$$A_b = (Other\_Capability \neq 1) \quad (3)$$

$$B = ((Climb\_Inhibit = 1) \wedge (Up\_Separation + 100 > Down\_Seperation)) \vee ((Climb\_Inhibit \neq 1) \wedge (Up\_Separation > Down\_Seperation)) \quad (4)$$

$$C = (Own\_Tracked\_Alt < Other\_Tracked\_Alt) \wedge (Down\_Seperation < Positive\_RA\_Alt\_Thresh[Alt\_Layer\_Value]) \quad (5)$$

$$D = (Other\_Tracked\_Alt < Own\_Tracked\_Alt) \wedge (Up\_Seperation \geqslant Positive\_RA\_Alt\_Thresh[Alt\_Layer\_Value]) \quad (6)$$

For this program, we define MRs as follows.

(a) **MR1: Modification in** *Own_Tracked_Alt* **and** *Other_Tracked_Alt*
These two parameters are signed integers that indicate the current altitudes of the controlled aircraft and the intruder aircraft, respectively.
Suppose *Own_Tracked_Alt* and *Other_Tracked_Alt* in the source test case t_s are denoted as X_s^1 and X_s^2, respectively. Let us define $AVE_s = \left(X_s^1 + X_s^2\right)/2$. Then, the follow-up test case t_f is constructed by replacing X_s^1 and X_s^2 in t_s with X_f^1 and X_f^2 in the following ways, according to the output of t_s (denoted as O_s).
- If O_s = *UPWARD*, we should have $A \wedge B \wedge C$ to be true. From Eq. (5), $X_s^1 < X_s^2$. Then in t_f, set $X_f^1 = AVE_s - 1$ and $X_f^2 = AVE_s + 1$.
- If O_s = *DOWNWARD*, we should have $A \wedge \neg B \wedge D$ to be true. From Eq. (6), $X_s^2 < X_s^1$. Then in t_f, set $X_f^1 = AVE_s + 1$ and $X_f^2 = AVE_s - 1$.
- If O_s = *UNRESOLVEDWARD*, then in t_f, set $X_f^1 = X_f^2 = AVE_s$.

Obviously, the truth values of A and B are the same for t_s and t_f. Hence, the relation between *Own_Tracked_Alt* and *Other_Tracked_Alt* remains unchanged, and consequently, the truth values of C and D are unchanged. Therefore, the output of t_f (denoted as O_f) will be the same as O_s.

(b) **MR2: Modification in** *Up_Separation* **and** *Down_Separation*
These two parameters are signed integers that indicate the vertical separation between the two aircraft when they reach the closest point of approach if the controlled aircraft performs an upward and downward maneuver, respectively.
Suppose *Up_Separation* and *Down_Separation* in the source test case t_s are denoted as Y_s^1 and Y_s^2, respectively. Then, the follow-up test case t_f is constructed by replacing Y_s^1 and Y_s^2 in t_s with Y_f^1 and Y_f^2 in the following way:
- If O_s = *UPWARD*, then set $Y_f^1 = Y_s^1$ and $Y_f^2 < Y_s^2$. Then, the truth values of A, B and C are unchanged. Thus, $O_f = O_s$.
- If O_s = *DOWNWARD*, then set $Y_f^1 > Y_s^1$ and $Y_f^2 = Y_s^2$, such that $Y_f^1 + 100 \leqslant Y_f^2$ if the parameter *Climb_Inhibit* in t_s is 1; otherwise $Y_f^1 \leqslant Y_f^2$. Then, the truth values of A, B and D are unchanged. Thus, $O_f = O_s$.
- If O_s = *UNRESOLVEDWARD*, then set $Y_f^1 < Y_s^1$ and $Y_f^2 > Y_s^2$, such that the relation between Y_f^1 and $Y_f^2 - 100$ is the same as the relation between Y_s^1 and $Y_s^2 - 100$, if *Climb_Inhibit* in t_s is 1; otherwise, the relation between Y_f^1 and Y_f^2 is the same as the relation between Y_s^1 and Y_s^2. Then, according to Eqs. (1)–(6), we have $O_f = O_s$.

(c) **MR3: Modification in** *Alt_Layer_Value*
tcas has a built-in array *Positive_RA_Alt_Thresh*[4] = {400, 500, 640, 740}, which defines four threshold levels. The vertical separation between two aircraft at their closest point of approach is considered adequate if it is greater than the specified threshold level. The parameter *Alt_Layer_Value* is used to specify which threshold level is used for reference. Let Z_s denote *Alt_Layer_Value* in t_s. Then, the follow-up test case t_f is constructed by replacing Z_s in t_s with Z_f as follows:

[1] Examination is conducted on each selected command segment to check whether all "BLOCK" and "UNBLOCK" commands in this segment have operated on valid job pointers. If not, the current segment is discarded and another one is selected. Selection and examination is repeated until an eligible command segment is found to serve as the source command segment.

- If O_s = *UPWARD*, set $Z_f = Z_s + 1$ $(0 \leqslant Z_s < 3)$. Then, the truth values of *A*, *B* and *C* are unchanged. Thus, $O_f = O_s$.
- If O_s = *DOWNWARD*, set $Z_f = Z_s - 1$ $(0 < Z_s \leqslant 3)$. Then, the truth values of *A*, *B* and *D* are unchanged. Thus, $O_f = O_s$.
- If O_s = *UNRESOLVEDWARD*,
  - If in t_s, *Own_Tracked_Alt* ⩽ *Other_Tracked_Alt*, set $Z_f = Z_s - 1 (0 < Z_s \leqslant 3)$.
  - Otherwise, set $Z_f = Z_s + 1$ $(0 \leqslant Z_s < 3)$.

According to Eqs. (1)–(6), we have $O_f = O_s$.

4.1.3.5. Tot_info. Given a source test case having *n* tables $T_s = \{t_1^s, t_2^s, \ldots, t_n^s\}$. For each table $t_i^s \in T_s$, *tot_info* prints $(info)_i^s$, $(df)_i^s$ and q_i^s, where $(info)_i^s$, $(df)_i^s$ and q_i^s denote the Kullbacks information measure, the degree of freedom and the possibility density of χ^2 distribution of t_i^s, respectively. In addition, *tot_info* prints $(tot\_info)^s$, $(tot\_df)^s$ and q^s, where $(tot\_info)^s$ and $(tot\_df)^s$ are the summaries of all $(info)_i^s$ and all $(df)_i^s$, respectively, and q^s is the possibility density of χ^2 distribution calculated with $(tot\_info)^s$ and $(tot\_df)^s$. Let us denote the *m* tables in the follow-up test case as $T_f = \left\{t_1^f, t_1^f, \ldots, t_m^f\right\}$. For t_f, the printed results are denoted as $(info)_i^f$, $(df)_i^f$, q_i^f, $(tot\_info)^f$, $(tot\_df)^f$ and q^f. The MRs for *tot_info* are as follows:

(a) **MR1: Duplication of the whole input file**
In this MR, T_f is constructed by duplicating all $t_s^i \in T_s$, that is, $T_f = T_s \cup T_s$. Then, we have $(tot\_info)_f = 2 * (tot\_info)_s$ and $(tot\_df)_f = 2*(tot\_df)_s$.

(b) **MR2: Duplication of one table**
For an arbitrarily chosen table $t_i^s \in T_s$, suppose t_i^s has r_i^s rows and c_i^s columns. In T_f, t_i^f is defined to have r_i^f rows and c_i^f columns such that $r_i^f = 2 * r_i^s$ and $c_i^f = c_i^s$. And the content from the $(r_i^s + 1)$th row to the $(2 * r_i^s)$th row in t_i^f is the duplicate of its first r_i^s rows. As a consequence, we have $(info)_i^f = 2 * (info)_i^f$ and $(df)_i^f = (df)_i^s + r_i^s * (c_i^s - 1)$.

(c) **MR3: Scaling the value of each element in one table**
For an arbitrarily chosen table $t_i^s \in T_s$, T_f is constructed by defining t_i^f such that each value in t_i^f is the corresponding value in t_i^s multiplied by *k*. Then, we have $(info)_i^f = k * (info)_i^s$ and $(df)_i^f = (df)_i^s$.

These seven programs also have the oracle problem. For example, in *replace*, it is difficult to automatically check whether all the substrings that match the regular expression have been picked up. The versions of all the seven programs used in our experiments are 2.0, which are downloaded from SIR [28].

4.2. Source test suite generation

For *grep*, even though the SIR has 807 test cases, we do not utilize them because only quite a small portion of them are "eligible source test cases" with respect to our MRs. In other words, most of these test cases do not contain specific substrings such that our MRs are applicable. Therefore, in order to have sufficient source test cases for our experiments, we use a test pool with 171,634 random test cases, from one of our previous studies. Among them, 2982 test cases are eligible source test cases for MR1, 5003 for MR2, and 2084 for MR3.

For *SeqMap*, since there is no existing test suite, we use our randomly generated test cases from a previous study [21] as the source test cases. This source test suite contains 300 test cases. Each test case consists of a set of short strings *P*, a long reference string *t* and the maximum number of mismatches *e*. *t* and *e* are the same in all the 300 test cases; while *P* varies in different test cases.

Table 2
Number of source test cases.

Program	Number
grep	2982 (MR1) 5003 (MR2) 2084 (MR3)
SeqMap	300
print_tokens	4130
print_tokens2	4115
replace	5542
schedule	2650
schedule2	2710
tcas	1608
tot_info	1052

For the seven programs in *Siemens Suite*, we simply adopt the "universe" test plans provided by SIR, as our source test suites. The numbers of the source test cases for all the nine programs are listed in Table 2.

4.3. Mutant generation

SIR has several released mutants for *Siemens Suite* and *grep*. However, many of them are equivalent mutants with respect to our MRs, and hence no violation will be expected. In the previous studies of SBFL with *e_slice*, mutants with no failure revealed were excluded. Likewise, mutants with no violation are also excluded in our investigation. Therefore, for the *Siemens Suite* and *grep* programs, apart from their feasible mutants, we randomly generate 300 extra mutants. For *SeqMap*, since there are no existing mutants available, we generate 300 mutants randomly.

Our mutant generation focuses on the non-omission and first-order faults (that is, single-fault mutants). Multiple-faults will be investigated in future. Two types of mutant operators are used: statement mutation and operator mutation. For statement mutation, either a "*continue*" statement is replaced with a "*break*" statement (and vice versa), or the label of a "*goto*" statement is replaced with another valid label. For operator mutation, an arithmetic (or a logical) operator is substituted by a different arithmetic (or logical) operator. To generate a mutant, our tool first randomly selects a line in the relevant part of the source code, and then searches systematically for possible locations where a mutant operator can be applied. One of these mutant operators is then selected randomly and applied randomly to one of the possible locations.

For each program, the mutants which could not be compiled successfully are first excluded. Also excluded include the mutants with an exceptional exit (crash-failure) because their coverage information could not be correctly collected, as well as the mutants without any violated metamorphic test group. The numbers of the feasible mutants for each program in each MR are shown in Table 3.

4.4. Measurement

Obviously, our proposed approach is intuitively appealing and conceptually feasible. But it is not clear whether it is practically effective or not. Thus, we applied our approach on the investigated programs. In our experiments, we have chosen three SBFL techniques, namely Tarantula [6], Jaccard [7] and Ochiai [10], whose risk evaluation formulas are listed in Table 4.

It is widely accepted that SBFL techniques are effective for programs with a test oracle. Thus, it is natural to investigate whether our proposed approach can deliver a similar performance as the existing SBFL techniques for the situation with test oracle. Hence, we have applied the conventional SBFL techniques with *e_slice* in our experiments as a benchmark. In essence, we are interested in seeing whether there is a significant impact on the effectiveness when the conventional *e_slice* is replaced by *e_mslice* and the test

Table 3
Number of mutants.

Program	MR1	MR2	MR3
grep	78	32	36
SeqMap	49	27	21
print_tokens	24	32	21
print_tokens2	31	27	21
replace	49	53	42
schedule	31	27	40
schedule2	36	37	21
tcas	16	26	27
tot_info	21	27	28

Table 4
Investigated formulas.

Name	Formula expression $R(A)$
Tarantula	$\frac{a_{ef}}{a_{ef}+a_{nf}} / \left(\frac{a_{ef}}{a_{ef}+a_{nf}} + \frac{a_{ep}}{a_{ep}+a_{np}}\right)$
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$
Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$

result of *failed* or *passed* is replaced by the metamorphic test result of *violation* or *non-violation*. If there is no significant effectiveness difference, our approach can be considered as a feasible approach in practice. Overall, we conduct SBFL under the following four scenarios:

- MS: SBFL using *e_mslice* with all eligible metamorphic test groups
- S-ST: SBFL using *e_slice* with all eligible source test cases
- S-FT: SBFL using *e_slice* with all eligible follow-up test cases
- S-AT: SBFL using *e_slice* with all eligible source and eligible follow-up test cases

Two interesting observations for these four scenarios are:

1. The number of *e_slice* used in S-ST or S-FT is the same as the number of *e_mslice* used in MS, but the number of *e_slice* used in S-AT is doubled. Therefore, S-ST, S-FT and MS have the same amount of raw data for the risk evaluation formulas, and hence their performance evaluation is expected to have the same degree of reliability.
2. MS and S-AT have the same number of test executions, which is twice the number of test executions of S-ST and S-FT. Thus the program execution overheads of MS and S-AT are the same, while the program execution overheads of S-ST and S-FT are also similar, but about half of those for either MS or S-AT.

As a reminder, S-ST, S-FT and S-AT require a test oracle. Similar to some previous studies, we use the non-mutated versions as the assumed test oracles in these three scenarios in our experiments. Furthermore, we adopt the widely used *EXAM* score as the effectiveness metric, which is the percentage of code that **needs** to be examined before the faults are identified [11]. Obviously, a lower *EXAM* score indicates a better performance. For statements with the same risk values, we rank them according to their original order in the source code. It should be noted that there are other ordering methods [30,11]. All experiments are conducted on a cluster of 64-bit Intel Clovertown systems running CentOS 5, and the statement coverage is collected by using *gcov*.

5. Experimental results

In our experiments, we investigate the effectiveness of our approach using two methods: the *EXAM* score distribution and the Wilcoxon-Signed-Rank Test.

5.1. EXAM score distribution

We utilize the line diagram to present the *EXAM* distribution of MS, S-ST, S-FT and S-AT, for all possible combinations of program and risk evaluation formula. The results are presented in Fig. 4. Each graph collects and summarizes the results over all the three MRs, for a particular combination of program and formula. In these figures, the horizontal axis means the *EXAM* score, while the vertical axis means the cumulative percentage of the mutants whose *EXAM* scores are less than or equal to the corresponding *EXAM* score. Obviously, the faster that the graph increases and reaches 100% of mutants, the better performance the technique has.

Generally speaking, we can observe from these graphs that the *EXAM* distributions are very similar in all these four scenarios. For example, in program *replace*, these four scenarios have almost identical performance. In program *print_tokens2*, the curves of MS are inter-crossing with the other three curves, such that MS performs slightly worse at the low *EXAM* score, but its curves increase sharply with the increase of the *EXAM* scores, and finally exceeds the other three curves to reach 100% of mutants at a smaller *EXAM* score. In *grep*, the curves of MS are above the other three scenarios, until they all achieve 100%.

In summary, the *EXAM* score distribution does not show any visually significant difference between our approach and the existing techniques.

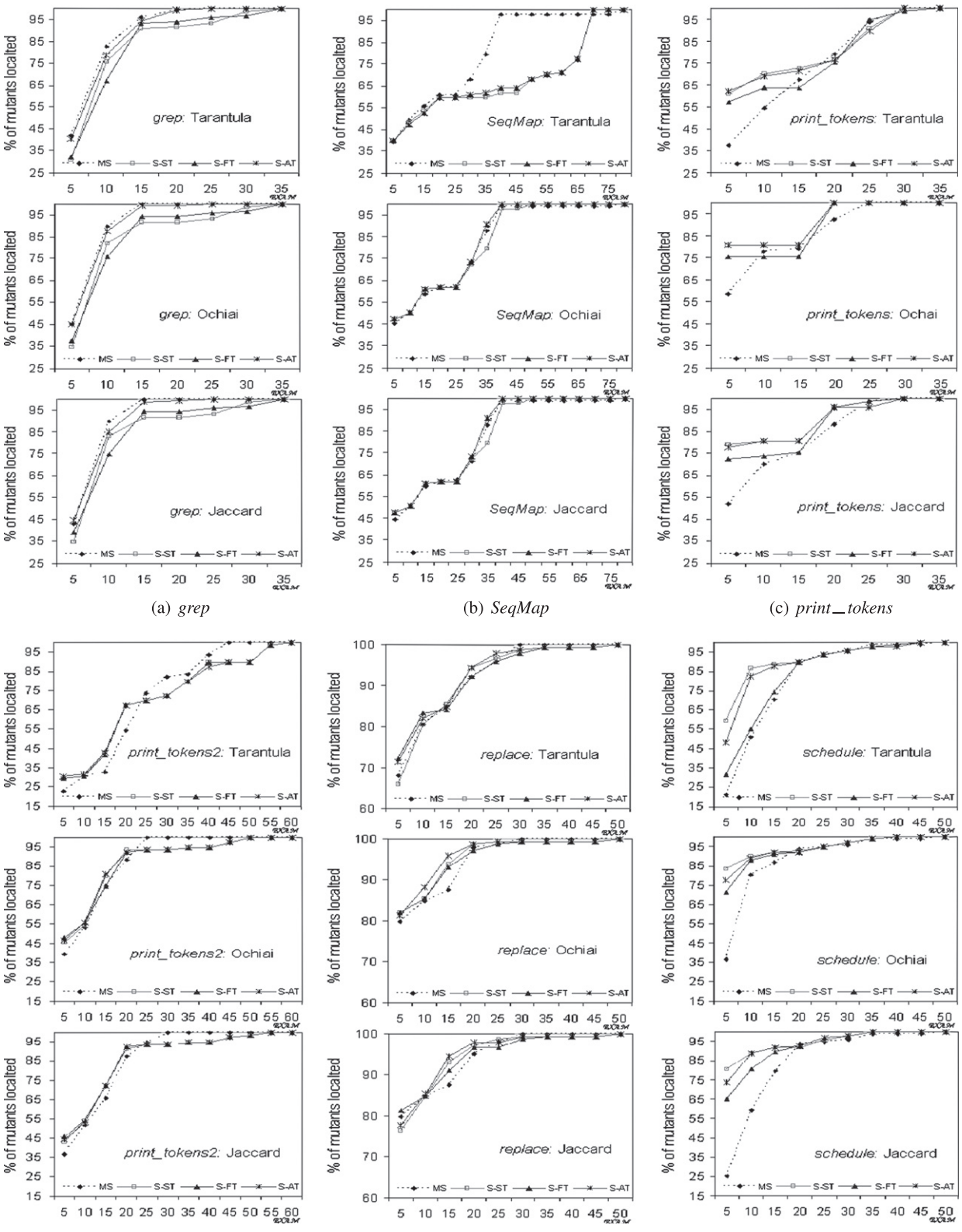
5.2. Statistical comparison

The *EXAM* score distribution only provides a visual comparison and is not rigorous enough. Thus, we further conduct a more rigorous and scientific comparison, namely, the paired Wilcoxon-Signed-Rank Test. The paired Wilcoxon-Signed-Rank test is a non-parametric statistical hypothesis test that makes use of the sign and the magnitude of the rank of the differences between pairs of measurements $F(x)$ and $G(y)$, which do not follow a normal distribution [31]. At the given significant level σ, there are both 2-tailed p-value and 1-tailed p-value which can be used to obtain a conclusion.

For the 2-tailed p-value, if $p \geqslant \sigma$, the null hypothesis H_0 that $F(x)$ and $G(y)$ are not significantly different is accepted; otherwise, the alternative hypothesis H_1 that $F(x)$ and $G(y)$ are significantly different is accepted. For 1-tailed p-value, there are two cases, the lower case and the upper case. In the lower case, if $p \geqslant \sigma$, H_0 that $F(x)$ does not significantly tend to be greater than the $G(y)$ is accepted; otherwise, H_1 that $F(x)$ significantly tends to be greater than the $G(y)$ is accepted. And in the upper case, if $p \geqslant \sigma$, H_0 that $F(x)$ does not significantly tend to be less than the $G(y)$ is accepted; otherwise, H_1 that $F(x)$ significantly tends to be less than the $G(y)$ is accepted.

In our experiments, we conducted three paired Wilcoxon-Signed-Rank tests, for MS vs. S-ST, MS vs. S-FT as well as MS vs. S-AT, using both the 2-tailed and 1-tailed checking, at the σ level of 0.05. Translated into our context, for a given combination of program and formula, the list of measurements of $F(x)$ is the list of the *EXAM* scores for all the mutants with MS; while the list of measurements of $G(y)$ is the list of the *EXAM* scores for all the mutants with S-ST, S-FT and S-AT, respectively. Therefore, in the 2-tailed test, H_0 being accepted means MS has SIMILAR performance as S-ST, S-FT or S-AT, at the significant level of 0.05. And in the 1-tailed test (lower), H_1 being accepted means MS has WORSE performance than S-ST, S-FT or S-AT, at the significant level of 0.05. Finally, in the 1-tailed test (upper), H_1 being accepted means MS has BETTER performance than S-ST, S-FT or S-AT, at the significant level of 0.05.

We use OriginPro 8.1 developed by OriginLab for this analysis. Tables 5–7 present the results for the comparison between MS and S-ST, MS and S-FT and MS and S-AT, respectively, where T, O and J mean formula Tarantula, Ochiai and Jaccard, respectively.

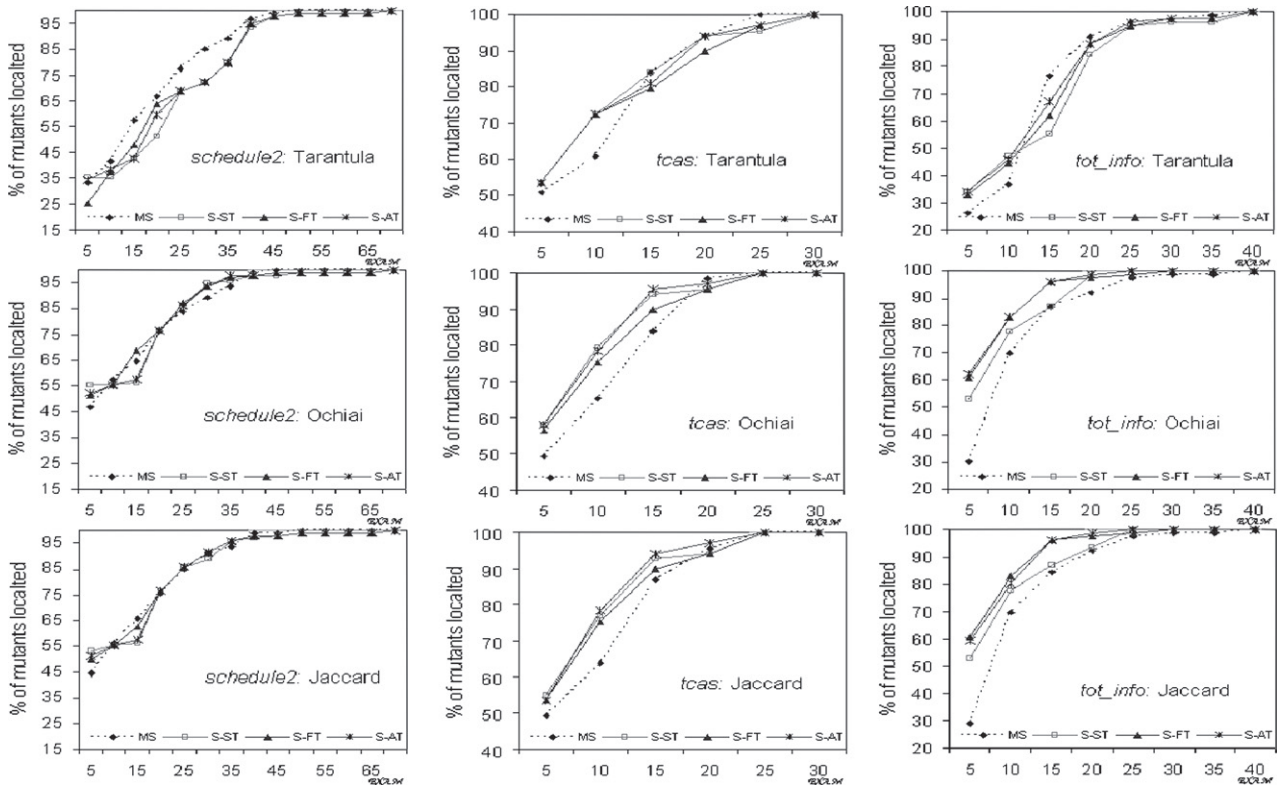


(a) *grep* (b) *SeqMap* (c) *print_tokens*

(d) *print_tokens2* (e) *replace* (f) *schedule*

Fig. 4. Distribution of *EXAM* scores.

Table 8 summarizes the numbers of SIMILAR, BETTER and WORSE cases in Tables 5–7. It shows that except for the comparison of MS vs. S-FT using Ochiai and Jaccard, in all other cases, BETTER and SIMILAR results occur more frequently than WORSE results. Overall, we have over 50% BETTER and SIMILAR results. Thus, the performance of MS is statistically comparable to the performance of other three scenarios where the test oracle is assumed.



(g) *schedule2* (h) *tcas* (i) *tot_info*

Fig. 4. (*continued*)

Table 5
MS vs. S-ST.

Program		2-Tailed	1-Tailed (lower)	1-Tailed (upper)	Conclusion
grep	T	0.00748	0.99628	0.00374	BETTER
	O	0.10282	0.9488	0.05141	SIMILAR
	J	0.21888	0.89093	0.10944	SIMILAR
SeqMap	T	1.46218E−06	1	7.3109E−07	BETTER
	O	0.99891	0.50164	0.49945	SIMILAR
	J	0.90321	0.54948	0.45161	SIMILAR
print_tokens	T	0.00659	0.00329	0.99677	WORSE
	O	8.9407E−08	4.47035E−08	1	WORSE
	J	1.41342E−05	7.06711E−06	0.99999	WORSE
print_tokens2	T	0.96774	0.51724	0.48387	SIMILAR
	O	0.15971	0.07986	0.92055	SIMILAR
	J	0.16053	0.08027	0.92013	SIMILAR
replace	T	0.22033	0.89027	0.11017	SIMILAR
	O	0.47632	0.23816	0.76255	SIMILAR
	J	0.30655	0.15328	0.84724	SIMILAR
schedule	T	4.32323E−07	2.16161E−07	1	WORSE
	O	2.98941E−09	1.4947E−09	1	WORSE
	J	5.61122E−10	2.80561E−10	1	WORSE
schedule2	T	0.000970491	0.99952	0.000485246	BETTER
	O	0.19735	0.09867	0.90206	SIMILAR
	J	0.22816	0.11408	0.88676	SIMILAR
tcas	T	0.13688	0.06844	0.93253	SIMILAR
	O	4.1329E−07	2.06645E−07	1	WORSE
	J	3.91155E−08	1.95578E−08	1	WORSE
tot_info	T	0.04193	0.97921	0.02097	BETTER
	O	0.00168	0.000837554	0.99918	WORSE
	J	0.00187	0.000933166	0.99909	WORSE

Table 6
MS vs. S-FT.

Program		2-Tailed	1-Tailed (lower)	1-Tailed (upper)	Conclusion
grep	T	2.66301E−07	1	1.33151E−07	BETTER
	O	0.000114414	0.99994	5.72071E−05	BETTER
	J	0.000238973	0.99988	0.000119486	BETTER
SeqMap	T	0.000015654	0.99999	7.82698E−06	BETTER
	O	0.45093	0.22546	0.77556	SIMILAR
	J	0.34456	0.17228	0.82859	SIMILAR
print_tokens	T	0.12978	0.06489	0.93588	SIMILAR
	O	0.01372	0.00686	0.99334	WORSE
	J	0.00211	0.00105	0.99897	WORSE
print_tokens2	T	0.44396	0.22198	0.77882	SIMILAR
	O	0.38151	0.19076	0.80995	SIMILAR
	J	0.13207	0.06604	0.9343	SIMILAR
replace	T	0.22251	0.88918	0.11125	SIMILAR
	O	0.6055	0.30275	0.69799	SIMILAR
	J	0.6538	0.3269	0.67387	SIMILAR
schedule	T	0.01871	0.00935	0.99076	WORSE
	O	6.60245E−09	3.30123E−09	1	WORSE
	J	4.05146E−10	2.02573E−10	1	WORSE
schedule2	T	0.000702801	0.99965	0.000351401	BETTER
	O	0.00584	0.00292	0.99712	WORSE
	J	0.12099	0.06049	0.94005	SIMILAR
tcas	T	0.96421	0.48211	0.52065	SIMILAR
	O	0.000546162	0.000273081	0.99973	WORSE
	J	0.000173005	8.65023E−05	0.99992	WORSE
tot_info	T	0.11317	0.94382	0.05659	SIMILAR
	O	7.91943E−08	3.95971E−08	1	WORSE
	J	1.43003E−07	7.15017E−08	1	WORSE

Table 7
MS vs. S-AT.

Program		2-Tailed	1-Tailed (lower)	1-Tailed (upper)	Conclusion
grep	T	0.03322	0.98347	0.01661	BETTER
	O	0.98409	0.50875	0.49204	SIMILAR
	J	0.87375	0.43688	0.5639	SIMILAR
SeqMap	T	1.65726E−05	0.99999	8.28628E−06	BETTER
	O	0.14614	0.07307	0.92731	SIMILAR
	J	0.13568	0.06784	0.93252	SIMILAR
print_tokens	T	0.0039	0.00195	0.99809	WORSE
	O	5.58794E−08	2.79397E−08	1	WORSE
	J	1.31834E−06	6.5917E−07	1	WORSE
print_tokens2	T	0.84935	0.42468	0.57639	SIMILAR
	O	0.18764	0.09382	0.90664	SIMILAR
	J	0.11254	0.05627	0.94403	SIMILAR
replace	T	0.68337	0.34168	0.65919	SIMILAR
	O	0.19999	0.1	0.90039	SIMILAR
	J	0.1096	0.0548	0.94544	SIMILAR
schedule	T	2.28286E−06	1.14143E−06	1	WORSE
	O	5.73627E−09	2.86814E−09	1	WORSE
	J	3.40643E−10	1.70322E−10	1	WORSE
schedule2	T	0.0001375	0.99993	6.87502E−05	BETTER
	O	0.13507	0.06754	0.93303	SIMILAR
	J	0.14197	0.07098	0.9296	SIMILAR
tcas	T	0.14073	0.07036	0.93062	SIMILAR
	O	3.70874E−07	1.85437E−07	1	WORSE
	J	5.57982E−11	2.78991E−11	1	WORSE
tot_info	T	0.38748	0.80732	0.19374	SIMILAR
	O	9.84774E−08	4.92387E−08	1	WORSE
	J	7.68342E−06	3.84171E−06	1	WORSE

5.3. Real-life faults in Siemens Suite

A by-product of our experiments is the identification of two real-life faults in the *Siemens Suite*, despite these programs having been extensively used and tested. This is understandable as metamorphic testing provides testing from a new perspective that has not been considered in the previous testing of the *Siemens Suite*. Hence, it is worthwhile to investigate the role and impact

Table 8
Summarized results.

Comparison		BETTER	WORSE	SIMILAR
MS vs. S-ST	T	4	2	3
	O	0	4	5
	J	0	4	5
MS vs. S-FT	T	3	1	5
	O	1	5	3
	J	1	4	4
MS vs. S-AT	T	3	2	4
	O	0	4	5
	J	0	4	5

of metamorphic testing as a test case generation methodology in future studies.

1. **Fault 1:** The first fault is at line 214 of the program *schedule*, in function: "void *upgrade_process_prio*(int *prio*, float *ratio*)". This function is for command "2 *prio ratio*", which upgrades a job from P_{prio} to its next higher priority. After processing the upgrade, the priority of the specified job should be increased by 1. But in line 214, the code is: "*proc* → *priority* = *prio*", rather than the correct "*proc* → *priority* = *prio* + 1".
2. **Fault 2:** The second fault is in the program *print_tokens*. In *print_tokens* and *print_tokens2*, there is a buffer with a length of 80 characters, to store the read-in tokens. However in *print_tokens*, even if a token is shorter than 80 characters, it also may be read and printed incorrectly. This is due to the way that *print_tokens* deals with the blanks.
   According to the specification, when reading in a token, these two programs should ignore all irrelevant blanks. In *print_tokens2*, before storing a valid token into the buffer, the program will first delete all irrelevant blanks. However, in *print_tokens*, a token will be stored into the buffer with its neighbor blanks. Removal of the blanks is performed afterward. Therefore, in *print_tokens*, the blanks occupy some space in the buffer which is of a fixed length of 80 characters, and hence, a valid token may only be partially stored by mistake.

5.4. Summary

According to the above analysis, we have the following conclusions.

1. The experimental data are very positive that our approach is not only conceptually feasible, but also has very similar performance as the conventional SBFL techniques, of which the performance has been generally regarded as effective by the testing community.
2. Despite the investigated programs being extensively used and well tested, the identification of two real-life faults in our study is a strong evidence that metamorphic testing provides a new perspective of generating test cases that can supplement other commonly used test case generation techniques. This further supports a conjecture in our recent study that diversity plays a key role in the fault detection effectiveness of test cases [32].

6. Threats to validity

6.1. External validity

The primary threat to the external validity is the generalizability of our results acquired from the nine test objectives, with only three MRs for each. It is well known that the seven programs in *Siemens Suite* are small in size. Although *grep* and *SeqMap* are considerably larger than these seven programs, they are still not very large-sized programs. Thus it is worthwhile to use more large-sized programs to further validate the effectiveness of our approach of using *e_mslice* in SBFL.

Another threat is the type of mutants (effectively, the type of faults) used in our study. Even though these mutants are randomly generated, each mutant contains exactly one fault and the types of faults are also restricted. Actually, our approach can be extended to programs with multiple faults. If there is a way (e.g., the fault focused approach in [33]) to segregate or cluster violated executions together such that violated test groups in each cluster are related to the same fault, then these relevant violated test groups along with some non-violated test groups can be used to locate a specific fault.

Besides the effectiveness of the MRs is not investigated in this study. We understand that identifying effective MRs is not a trivial task for some programs. However, there are recent studies demonstrating that in most cases, the identification of MRs are quite easy. Some empirical evidence shows that after a brief general training, testers can properly define MRs and effectively apply MT on the target programs [17,18].

Nevertheless, even if the selected MRs are not effective, our approach is still of significant impact because it provides a solution for programs without oracle, on which the conventional SBFL cannot be applied.

6.2. Internal validity

The primary threat to the internal validity involves the correctness of our experiment framework which includes the generation of source and follow-up test cases, the generation of mutants, execution of these mutants with both source and follow-up test cases, as well as examination of the outputs of source and follow-up test cases against the corresponding MR. Our experimental framework is quite different from the conventional SBFL using *e_slice*. Thus the existing test framework from SIR cannot be adopted completely. In order to assure the quality of our experimentation, we have conducted a very thorough unit testing at each implementation step and a comprehensive functional testing at the system level.

6.3. Construct validity

The primary threat to the construct validity is the use of the *EXAM* score as a measure of the effectiveness of a SBFL technique. However, this score has been extensively used [34,8,9,11], so the threat is acceptably mitigated.

7. Conclusion

SBFL has been extensively investigated, and numerous investigations have proposed various effective SBFL techniques. However, all these techniques have assumed that there exists a test oracle. Since in practice many application domains do not have the test oracle, this has severely restricted the applicability of the existing SBFL techniques. Recently, MT has been proposed to alleviate the oracle problem. Thus, it is natural to investigate how MT could be used to alleviate the oracle problem in SBFL.

In this paper, we propose a novel concept, *metamorphic slice*, based on the integration of metamorphic relations and slices. In our proposal, for any existing SBFL techniques, the role of *slice* is replaced by that of *metamorphic slice*, the role of an individual test case is replaced by that of a metamorphic test group, and the test result of *failure* or *pass* of an individual test case is replaced by the test result of *violation* or *non-violation* of a metamorphic test group.

With these one-to-one replacements, we could then construct the metamorphic versions for the existing SBFL techniques.

An experimental study on nine programs of varying program sizes has been conducted to investigate the performance of our proposed approach and see whether it is practically effective or not. From the results, we can conclude that our approach has delivered a performance level very similar to that achieved by the conventional SBFL techniques for the situation with test oracle, which has been generally regarded as effective and satisfactory by the testing community. In future, we will conduct additional experimental studies on more large-sized programs, more types of faults, as well as programs with multiple-faults. The effectiveness of different MRs, and how to select better MR to provide a better fault localization performance, will also be studied.

This study only addresses one application of metamorphic slices. Hence, one future research direction is to revisit the applications of conventional slices which require the existence of test oracles and to investigate how to use metamorphic slices instead, so as to alleviate the oracle problem. Furthermore, since all previously developed types of slices are variable based and our notion of metamorphic slice is based on a different perspective, namely, property, we believe that such a distinction between the conventional slice and the metamorphic slice will provide insights into new research directions.

Acknowledgements

This project is partially supported by Australian Research Council Discovery Project (DP120104773), the National Natural Science Foundation of China (90818027, 60873050), and the USA National Science Foundation (NSF DUE-1023071).

References

[1] X.Y. Xie, W.E. Wong, T.Y. Chen, B.W. Xu, Spectrum-based fault localization: testing oracles are no longer mandatory, in: Proceedings of the 11th International Conference on Quality Software (QSIC), Madrid, Spain, 2011, pp. 1–10.
[2] J.S. Collofello, S.N. Woodfield, Evaluating the effectiveness of reliability-assurance techniques, Journal of Systems and Software 9 (1989) 191–195.
[3] W.E. Wong, T. Sugeta, Y. Qi, J.C. Maldonado, Smart debugging software architectural design in SDL, Journal of Systems and Software 76 (2005) 15–28.
[4] R. Abreu, P. Zoeteweij, A.J.C. van Gemund, On the accuracy of spectrum-based fault localization, in: Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION), Windsor, UK, 2007, pp. 89–98.
[5] T. Reps, T. Ball, M. Das, J. Larus, The use of program profiling for software maintenance with applications to the year 2000 problem, ACM SIGSOFT Software Engineering Notes 22 (1997) 432–449.
[6] J.A. Jones, M.J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the 24th International Conference on Software Engineering (ICSE), Florida, USA, 2002, pp. 467–477.
[7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: problem determination in large, dynamic internet services, in: Proceedings of the 32th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Washington DC, USA, 2002, pp. 595–604.
[8] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, M.I. Jordan, Scalable statistical bug isolation, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Chicago, USA, 2005, pp. 15–26.
[9] C. Liu, X. Yan, L. Fei, J. Han, S.P. Midkiff, SOBER: statistical model-based bug localization, IEEE Transactions on Software Engineering 32 (2006) 831–848.
[10] R. Abreu, P. Zoeteweij, A.J.C. Gemund, An evaluation of similarity coefficients for software fault localization, in: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC), Riverside, USA, 2006, pp. 39–46.
[11] W.E. Wong, V. Debroy, B. Choi, A family of code coverage-based heuristics for effective fault localization, Journal of Systems and Software 83 (2010) 188–208.
[12] A. Zeller, Isolating cause-effect chains from computer programs, in: Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), South Carolina, USA, 2002, pp. 1–10.
[13] X.Y. Xie, J.W.K. Ho, C. Murphy, G. Kaiser, B.W. Xu, T.Y. Chen, Application of metamorphic testing to supervised classifiers, in: Proceedings of the 9h International Conference on Quality Software (QSIC), Jeju, Korea, 2009, pp. 135–144.
[14] X.Y. Xie, J.W.K. Ho, C. Murphy, G. Kaiser, B.W. Xu, T.Y. Chen, Testing and validating machine learning classifiers by metamorphic testing, Journal of Systems and Software 84 (2011) 544–558.
[15] M.J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi, An empirical investigation of the relationship between spectra differences and regression faults, Software Testing Verification and Reliability 10 (2000) 171–194.
[16] T.Y. Chen, F.-C. Kuo, T.H. Tse, Z.Q. Zhou, Metamorphic testing and beyond, in: Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice (STEP), Amsterdam, The Netherlands, 2003, pp. 94–100.
[17] P.F. Hu, Z. Zhang, W.K. Chan, T.H. Tse, An empirical comparison between direct and indirect test result checking approaches, in: Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA), Portland, USA, 2006, pp. 6–13.
[18] Z.Y. Zhang, W.K. Chan, T.H. Tse, P.F. Hu, Experimental study to compare the use of metamorphic testing and assertion checking, Journal of Software 20 (2009) 2637–2654.
[19] W.K. Chan, S.C. Cheung, K.R.P.H. Leung, A metamorphic testing approach for online testing of service-oriented software applications, International Journal of Web Services Research 4 (2007) 61–81.
[20] T.Y. Chen, F.-C. Kuo, Z.Q. Zhou, An effective testing method for end-user programmer, in: Proceedings of the 1st Workshop on End-User Software Engineering (WEUSE), St. Louis, Missouri, USA, 2005, pp. 1–5.
[21] T.Y. Chen, J.W.K. Ho, H. Liu, X.Y. Xie, An innovative approach for testing bioinformatics programs using metamorphic testing, BMC Bioinformatics 10 (2009) 24–35.
[22] T.Y. Chen, D.H. Huang, T.H. Tse, Z.Q. Zhou, Case studies on the selection of useful relations in metamorphic testing, in: Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JISIC), Madrid, Spain, 2004, p. 569583.
[23] T.Y. Chen, T.H. Tse, Z.Q. Zhou, Semi-proving: an integrated method for program proving, testing, and debugging, IEEE Transactions on Software Engineering 37 (2011) 109–125.
[24] M. Harman, R.M. Hierons, C. Fox, S. Danicic, J. Howroyd, Pre/Post conditioned slicing, in: Proceedings of the 9th International Conference on Software Maintenance (ICSM), Florence, Italy, pp. 138–147.
[25] R.M. Hierons, M. Harman, C. Fox, L. Ouarbya, M. Daoudi, Conditioned slicing supports partition testing, Software Testing Verification and Reliability 12 (2002) 23–28.
[26] D. Binkley, M. Harman, A survey of empirical results on program slicing, Advances in Computers 62 (2004) 105–178.
[27] SLOCCount <http://www.dwheeler.com/sloccount/sloccount.html>, 2004.
[28] SIR <http://sir.unl.edu/php/index.php>, 2005.
[29] H. Jiang, W.H. Wong, SeqMap: mapping massive amount of oligonucleotides to the genome, Bioinformatics 24 (2008) 2395–2396.
[30] W.E. Wong, T. Wei, Y. Qi, L. Zhao, A crosstab-based statistical method for effective fault localization, in: Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST), Lillehammer, Norway, 2008, pp. 42–51.
[31] G.W. Corder, D.I. Foreman, Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach, Wiley, 2009.
[32] T.Y. Chen, F.-C. Kuo, R.G. Merkel, T.H. Tse, Adaptive random testing: the ART of test case diversity, Journal of Systems and Software 83 (2010) 60–66.
[33] J.A. Jones, J.F. Bowring, M.J. Harrold, Debugging in parallel, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), New York, USA, 2007, pp. 16–26.
[34] J.A. Jones, M.J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE), Long Beach, USA, 2005, pp. 273–282.