PAPER

Effective Fault Localization Approach Using Feedback

Yan LEI[†a)], ***Student Member***, **Xiaoguang MAO**[†b)], ***Nonmember***, **Ziying DAI**[†], ***and*** **Dengping WEI**[†], ***Student Members***

SUMMARY At the stage of software debugging, the effective interaction between software debugging engineers and fault localization techniques can greatly improve fault localization performance. However, most fault localization approaches usually ignore this interaction and merely utilize the information from testing. Due to different goals of testing and fault localization, the lack of interaction may lead to the issue of information inadequacy, which can substantially degrade fault localization performance. In addition, human work is costly and error-prone. It is vital to study and simulate the pattern of debugging engineers as they apply their knowledge and experience to this interaction to promote fault localization effectiveness and reduce their workload. Thus this paper proposes an effective fault localization approach to simulate this interaction via feedback. Based on results obtained from fault localization techniques, this approach utilizes test data generation techniques to automatically produce feedback for interacting with these fault localization techniques, and then iterate this process to improve fault localization performance until a specific stopping condition is satisfied. Experiments on two standard benchmarks demonstrate the significant improvement of our approach over a promising fault localization technique, namely the spectrum-based fault localization technique.

key words: *fault localization, software debugging, program spectra, feedback*

1. Introduction

Due to the sheer size and complexity of software, it is almost impossible to produce faultless software. For these reasons, software debugging is employed to find and fix bugs, and so plays a vital role in improving software quality. However, debugging is one of the most time-consuming tasks in the development and maintenance of software [1]. A typical process of debugging consists of four steps: failure reproduction, fault localization, fault repair and repair verification, among which fault localization is usually a tedious and difficult step [2]. With the aim at decreasing the cost of debugging, many researchers try to optimize the process of fault localization and improve its performance [1]–[25].

Although fault localization has made great progress in recent years, the published results are not so satisfactory. Typically, the process of fault localization is as follows. First, a software debugging engineer uses some fault localization techniques to locate faults. Then, he/she analyzes and deduces the primary results obtained from these fault localization techniques. Next, based on the analysis and deduction, some new test cases are generated to feed back to fault localization techniques to verify and improve the primary results. The above steps would be iterated until faults' locations are identified. It can be seen that there exists an interaction between software debugging engineers and fault localization techniques. This interaction is of great importance to promote fault localization performance.

However, most fault localization approaches usually ignore this interaction [8]–[21]. They assume that test cases satisfying a certain test adequacy criterion can provide adequate information for fault localization, and there are no new test cases to provide required feedback [5]. In addition, the goal of testing is to reveal faults instead of locating faults. Due to different goals of testing and fault localization, test cases satisfying a certain test adequacy criterion may fail to offer adequate information to fault localization. As inadequate information can greatly influence fault localization performance, it is vital to generate new test cases via the interaction to address the issue of information inadequacy.

At present, some interactive fault localization approaches have been already presented that can interact with debugging engineers [22]–[25]. Because the interaction reflects the brainpower of debugging engineers, it is genuinely difficult for fault localization approaches to analyze and deduce like engineers. Therefore, these approaches usually focus on how to narrow down debugging engineers' attention to a smaller searching scope. The core work of analyzing and deducing faults' locations still strongly depends on the knowledge and experience of debugging engineers, which incurs the intensive workload of debugging engineers in locating faults. To whatever extent, it is essential to study and simulate the pattern of debugging engineers as they apply their knowledge and experience to this interaction. If we can simulate part of or even the whole interaction process, fault localization performance can be improved and the burden on debugging engineers' shoulder can be further alleviated.

From what has been discussed above, this paper proposes an effective fault localization approach using feedback to simulate the interaction between debugging engineers and fault localization. This approach iteratively analyzes and deduces the results obtained from fault localization techniques, and produces feedback for these fault localization techniques to gradually verify and improve these results. The feedback here is automatically provided by new test cases. New test cases are generated by test data generation techniques to cover current most suspicious state-

Manuscript received March 14, 2012.
Manuscript revised May 24, 2012.

[†]The authors are with School of Computer, National University of Defense Technology, 410073, Changsha, China.
a) E-mail: yanlei@nudt.edu.cn
b) E-mail: xgmao@nudt.edu.cn (Corresponding author)

DOI: 10.1587/transinf.E95.D.2247

ments extracted from current fault localization result. With the help of new test cases, the fault localization technique obtains more useful information and its performance would be improved. In essence, our approach, to some extent, imitates the way of debugging engineers as they apply their knowledge and experience to fault localization. In order to fully demonstrate the effectiveness of our approach, we apply our approach to a promising fault localization technique, namely spectrum-based fault localization (SFL) [9]. Typically, SFL exploits the correlations between program entities and program failures by statistically analyzing coverage information. SFL has been widely studied and used in the field of fault localization [2]–[5], [8]–[21], and the prior research [2], [3] has empirically proven that SFL yields a promising ability in locating faults.

In this paper, our approach is applied to three representative ranking metrics of SFL, namely Ochiai [9], Jaccard [10] and Tarantula [11]. The experimental study is conducted on two standard benchmarks, the *Siemens suite* and *Space* [27], with 154 faulty versions. The experimental results demonstrate the significant improvement of our approach over the three typical ranking metrics of SFL.

The main contribution of this paper can be summarized as:

(1) A new approach is proposed to simulate the interaction between debugging engineers and fault localization. The strength of our approach is validated analytically and empirically.

(2) The effectiveness of our approach is systematically studied via the experimental study conducted on three typical ranking metrics of SFL (Ochiai, Jaccard and Tarantula) and representative standard benchmarks (the *Siemens suite* and *Space*), providing an initial quantification of the benefits to applying the proposed approach.

(3) Our findings suggest that test cases from testing are likely to offer information that is inadequate for conducting efficient fault localization, and the proper simulation of the interaction between debugging engineers and fault localization has great potential for improving fault localization effectiveness.

The remainder of this paper is organized as follows. Section 2 introduces the background of spectrum-based fault localization. Section 3 details the problem of insufficient interactions and its solution. Section 4 describes the overview of our approach and an illustrative example. Section 5 presents empirical results of the proposed approach over SFL. Section 6 discusses related work. Finally, the conclusion is summarized in Sect. 7.

2. Background of SFL

Spectrum-based fault localization (SFL) [9] is a dynamic program analysis technique ranking program entities whose activity correlates most with the failures. SFL utilizes program spectra both from passed and failed runs. Passed runs are executions of a program that output as expected, whereas failed runs are executions of a program that output as unexpected. A program spectrum is a collection of data that provides a specific view on the dynamic behavior of software. The program spectrum is collected at run-time, and typically records the coverage information for program entities. There are various types of program entities, such as blocks, functions, branches, paths, etc. This study adopts the most widely used type of program entities, namely statements.

$$
\begin{array}{c}
\\
M\ runs
\end{array}
\begin{array}{c}
\begin{array}{cccc} N\ statements & & & \end{array} \\
\begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} \\
\begin{array}{cccc} s_1 & s_2 & \cdots & s_N \end{array}
\end{array}
\begin{array}{c}
\begin{array}{c} errors \end{array} \\
\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \\
\\
\end{array}
\begin{array}{c}
\\
\begin{bmatrix} t_1 \\ t_2 \\ \vdots \\ t_M \end{bmatrix} \\
\\
\end{array}
$$

Fig. 1 Input to SFL.

First, we assume that a program P comprises a set of statements $S = \{s_1, s_2, \ldots, s_N\}$ and runs against a set of test cases $T = \{t_1, t_2, \ldots, t_M\}$ that contains at least one failed test case (see Fig. 1). Hence, $|S| = N$ and $|T| = M$. The above matrix $M \times (N + 1)$ represents the input to SFL. An element x_{ij} is equal to 1 if statement s_j is covered by the execution of test run t_i, and 0 otherwise. The error vector e at the rightmost column of the matrix represents the test results. The element e_i is equal to 1 if test run t_i failed, and 0 otherwise. Except the error vector, the rest of the matrix is expressed in terms of matrix A. The i^{th} row of A shows whether a statement was covered by test run t_i. The j^{th} column of A indicates that statement s_j was covered by which test runs, and also represents the statement spectra of s_j.

SFL usually measures the suspiciousness of a statement to be faulty from the similarity between its statement spectra and error vector in the above matrix (see Fig. 1), and finally outputs a ranking list of all statements in descending order of suspiciousness. The similarity is quantified by ranking metrics. Ochiai [9], Jaccard [10] and Tarantula [11] are three typical ranking metrics of SFL, and their formulas are shown as follows:

$$S_{Ochiai}(s_j) = \frac{a_{11}(s_j)}{\sqrt{(a_{11}(s_j) + a_{01}(s_j)) * (a_{11}(s_j) + a_{10}(s_j))}} \quad (1)$$

$$S_{Jaccard}(s_j) = \frac{a_{11}(s_j)}{a_{11}(s_j) + a_{01}(s_j) + a_{10}(s_j)} \quad (2)$$

$$S_{Tarantula}(s_j) = \frac{(\frac{a_{11}(s_j)}{a_{11}(s_j)+a_{01}(s_j)})}{(\frac{a_{11}(s_j)}{a_{11}(s_j)+a_{01}(s_j)}) + (\frac{a_{10}(s_j)}{a_{10}(s_j)+a_{00}(s_j)})} \quad (3)$$

Where $a_{pq}(s_j) = |\{t_i \mid (x_{ij} = p) \wedge (e_i = q)\}|$, and $p, q \in \{0, 1\}$. The following shows the meaning of each variable.

(1) $x_{ij} = p$ denotes whether s_j was executed ($p = 1$) in the execution of t_i or not ($p = 0$).

(2) $e_i = q$ represents whether t_i was failed ($q = 1$) or not ($q = 0$).

(3) $a_{00}(s_j)$ denotes the number of passed test cases that does not execute s_j, and $a_{01}(s_j)$ represents the number of

failed test cases that does not execute s_j.

(4) $a_{10}(s_j)$ represents the number of passed test cases that execute s_j, and $a_{11}(s_j)$ represents the number of failed test cases that execute s_j.

(5) $S_{Ochiai}(s_j)$, $S_{Jaccard}(s_j)$ and $S_{Tarantula}(s_j)$ represent the suspiciousness of s_j computed by Ochiai, Jaccard and Tarantula respectively.

SFL is independent of any specific model of system and incurs low time and space overhead. Due to the feature of statistics and its simplicity, SFL is widely accepted and studied as a promising technique in the fault localization community [2]–[5], [8]–[21], and the research [2], [3] has also empirically proven that SFL has high effectiveness of locating faults. Therefore, our approach is applied to SFL to fully demonstrate its effectiveness.

3. Problems and Solution

3.1 Problems

This section uses the program P and the set of test cases T defined in Sect. 2. To explain the problems simply, we further assume that P contains only one faulty statement. Please note that the following analysis is also applicable to multiple faults because it is based on the principle of SFL that has been empirically proven to be effective in the context of multiple faults [4]. A metric named *failed probability* (referred as *FP*) is defined as follows:

$$FP(stm, T) = \frac{failedNum(stm, T)}{totalNum(stm, T)} \quad (4)$$

In Eq. (4), *stm* denotes a statement in the program P. The $failedNum(stm, T)$ represents the number of failed test cases in T that executed *stm*. The $totalNum(stm, T)$ is the number of test cases in T that executed *stm*. $FP(stm, T)$ represents the failed probability of all test cases covering *stm* in T. When T comprises all data in whole input space, $FP(stm, T)$ is named *global failed probability* (referred as $GFP(stm)$). $GFP(stm)$ indicates the failed probability of all test cases executing *stm* in whole input space. The set of statements whose *GFP* is lower than that of the faulty statement is called *affected set* (see Fig. 2).

The previous research [8] used the *FP* of a statement as an indication of how consistent the activity of the statement is with failures in a set of test cases. They investigated the relationship between the performance of SFL and fault consistency, and found that SFL assigns higher suspiciousness to the faulty statement when its *FP* is higher. In other words, when the activity of a statement is more consistent with failures, SFL would assign higher suspiciousness to the statement. This finding implies that a statement with a higher *FP* should be more suspicious to be faulty (referred as FL Rule), and FL Rule considerably impacts on SFL as it evaluates the suspiciousness of each statement. Therefore, it can be concluded that SFL has explicitly or implicitly adopted the idea of FL Rule [8]–[21].

Apparently, the effectiveness of FL Rule depends on

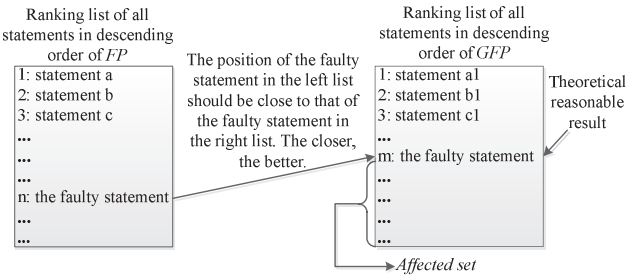

Fig. 2 Ranking lists of all statements in terms of *FP* and *GFP* respectively.

the set of test cases used because the values of *FP* are related to a set of test cases. Different sets of test cases, in general, are biased toward different requirements, such as different code coverage, and therefore create different effects on the effectiveness of FL Rule. It is vital to provide a criterion to assess and improve the adequacy of the set of test cases for fault localization. As *GFP* is acquired from the whole input space rather than a subset of the whole input space, it provides an unbiased and more comprehensive indication of how consistent the activity of a statement is with failures as compared to *FP*. This reveals that the place of the faulty statement in the ranking list of all statements in descending order of *GFP* is the theoretical reasonable position where it should stay. It requires the place of the faulty statement in the ranking list of all statements in descending order of *FP* should be close to its position in that ranking list in terms of *GFP* (See Fig. 2). Due to lower *GFP*, the statements in *affected set* should be less suspicious to be faulty compared with the faulty statement. Hence, the *FP* of the faulty statement should be higher than that of statements in *affected set*, which is the criterion that test cases for fault localization should meet.

However, the goal of testing is to reveal faults rather than locate faults. Test cases from testing should cover as many different statements as possible to reveal as many faults as possible. Apparently, these test cases are not designed for satisfying the aforementioned criterion on test cases for fault localization. This may lead to a higher *FP* that some statements in *affected set* obtain than the faulty statement. Consequently, it is more probable to judge these statements in *affected set* to be a fault compared with the faulty statement, which causes the effectiveness of FL Rule drastically decreases. In addition, as the whole input space is usually fairly large, it is unfeasible to obtain *GFP* by the exhaustive method. It is vital to utilize the interaction to address this inadequacy issue.

3.2 Solution

This section adopts the program P and set of test cases T defined in Sect. 3.1. Let $totalNum(stm, T) = K$, $failedNum(stm, T) = I$, $r = I/K$ and the number of test cases covering *stm* in whole input space be H. There is a new set of test cases T_{new} with $L = |T_{new}|$. We assume that

each test case in T_{new} would execute *stm* and $T_{new} \cap T = \phi$. For each of test cases covering *stm* in whole input space except T, we hypothesize its probability to be failed is q. T_{new} is added to T and the $FP(stm, T \cup T_{new}) = (K*r+L*q)/(K+L)$. In addition, $GFP(stm) = (K*r+(H-K)*q)/H$. Hence, the difference between $GFP(stm)$ and $FP(stm, T \cup T_{new})$ is computed as follow:

$$\begin{aligned} &|GFP(stm) - FP(stm, T \cup T_{new})| \\ &= |\frac{K*r+(H-K)*q}{H} - \frac{K*r+L*q}{K+L}| \\ &= |\frac{K*(H-K-L)*(r-q)}{H*(K+L)}| \end{aligned} \quad (5)$$

As H is the number of all test cases covering *stm* in whole input space, $(H-K-L) \geq 0$. Through increasing L, $(H-K-L)$ becomes lower while $(K+L)$ turns higher. Hence the difference between $GFP(stm)$ and $FP(stm, T \cup T_{new})$ could become smaller by increasing L (see Eq. (5)). It can be concluded that it is more probable to make the *FP* of a statement closer to its *GFP* by adding new test cases covering this statement to initial test cases.

From the above analysis, the solution of our approach is to extract current most suspicious statements from current results outputted by fault localization techniques, and feed back new test cases covering current most suspicious statements to these fault localization techniques to obtain new refined fault localization results. The above interaction would be iterated until the change of fault localization results is less than a setting threshold. During the constant interaction, new test cases would make the *FP* of current most suspicious statements closer to their *GFP*, especially reducing the *FP* of those current most suspicious statements belonging to *affected set*. In the following, three cases are used (See Fig. 3):

1) First of all, current most suspicious statement is the faulty statement. The *FP* of the faulty statement becomes closer to its *GFP* via new test cases.

2) Then, current most suspicious statement belongs to *affected set*. As mentioned in Sect. 3.1, this kind of statement should be ranked lower than the faulty statement. New test cases would reduce unusual high *FP* of current most suspicious statement to become closer to its *GFP*. As a result, it is more probable to make *FP* of this suspicious statement lower than that of the faulty statement.

3) Lastly, current most suspicious statement is not the faulty statement and does not belong to *affected set*. The *GFP* of this kind of statement is higher than or equal to that of the faulty statement. New test cases cannot make this kind of statements less suspicious than the faulty statement. However, due to high *GFP* of this statement, new test cases covering this statement are more likely to fail. Failed test cases should cover the faulty statement. Hence it is more probable to make the *FP* of the faulty statement closer to its *GFP*.

It can be seen that new test cases covering current most suspicious statements can make the places of the faulty statement of the two ranking lists in Fig. 2 become closer and closer. This promotes the effectiveness of FL Rule. It suggests that new test cases generated by the above solution can achieve two effects. One is to partially deduce and verify current fault localization result. The other is to improve current result according to the deduction and verification. The following experimental evaluation of the proposed approach also conforms to this conclusion.

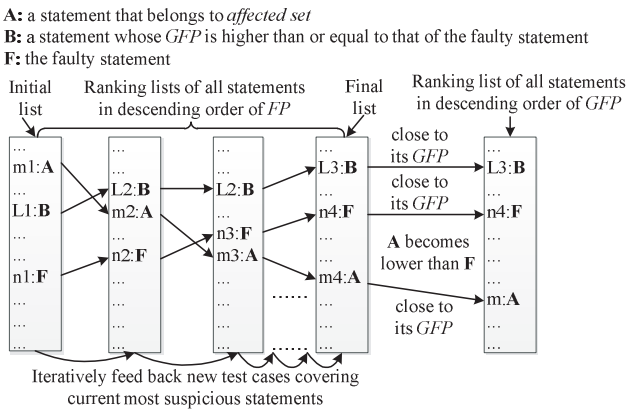

Fig. 3 Solution of our approach.

4. The Approach

4.1 Overview

As discussed in Sect. 3.1, SFL has explicitly or implicitly adopted the idea of FL Rule. Thus the application of our approach to SFL is implemented based on the solution described in Sect. 3.2. As shown in Fig. 4, there are three parts in our approach, namely SFL analyzer, most suspicious statements extractor and random data generator and data selector. The detailed steps of our approach are described as follows:

1) To begin with, the execution information of initial test case is inputted to SFL analyzer. It uses SFL to analyze the statement coverage and test results, and compute the suspiciousness of each statement. Finally, it outputs a ranking list of all statements in descending order of their suspiciousness.

2) Next, current ranking list of all statements is inputted

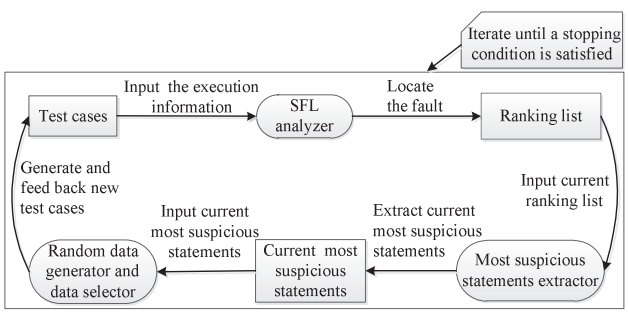

Fig. 4 Overview of the approach.

to most suspicious statements extractor. The extractor selects current most suspicious statements from current ranking list in terms of suspiciousness.

3) Furthermore, current most suspicious statements are inputted to random data generator and data selector. Random data generator generates a set of random data. In this set of data, test cases that cover any one of current most suspicious statements are chosen by data selector. These chosen test cases as the feedback are added to current test cases. Considering the cost and simplicity, our approach chooses random data generation technique. Note that other types of test data generation techniques, such as goal-oriented test generation and path-oriented test generation [26], can be also chosen by this step according to specific requirements.

4) Finally, the chosen test cases are feeding back to current test cases to constitute a new set of test cases. The execution information of the new set of test cases are inputted to SFL analyzer to output a new ranking list of all statements in descending order of suspiciousness. Step 2) to step 4) are iterated until the difference between two adjacent iterations' fault localization results is less than a setting threshold or it exceeds the maximum number of iterations according to new cost limit.

It is essential for our approach to set a proper value of the threshold. A high value of the threshold may cause that the iteration is too quickly terminated, and therefore an insufficient number of new test cases may be generated. That can restrict our approach to fully show its ability. In contrast, a low value of the threshold may cause that the iteration is too slowly terminated, and consequently an excessive number of new test cases may be produced. That can make the effectiveness of new test cases considerably decrease. The previous research [9] has found that the fault can be located by inspecting an average of the top 20% of statements in the ranking list given by SFL. Base on this finding, this study sets the threshold as the top 20% of statements in the ranking list[†]. In other words, if the top 20% of statements of the two adjacent iterations' ranking lists keep the same as each other, the iteration will be terminated.

SFL analyzer is implemented based on *Gcov* tool. *Gcov* tool is a test coverage program used in concert with *Gcov*. Ochiai [9], Jaccard [10] and Tarantula [11] are all implemented by SFL analyzer. Random data generator is implemented by utilizing random data generation algorithm of C language library. The statements are ranked in descending order of their suspiciousness assigned by the SFL. For the statements with the same assigned suspiciousness, we rank them according to their original order in the source code.

4.2 An Illustrative Example

This section illustrates an simple example to show just how our approach is to be applied. Figure 5 (a) presents a faulty program that contains a faulty statement s_7. Figure 5 (b) shows how our approach runs in one ranking metric of SFL. The cells below each statement indicate whether the statement was covered by the execution of a test case or not and the rightmost cells represent whether the execution of a test case is failed or not. The detailed description of the cells can refer to the input matrix of SFL presented in Fig. 1. As shown in Fig. 5 (b), based on the statement coverage and test results of initial five test cases, Ochiai outputs a ranking list of all statements in descending order of suspiciousness: $\{s_5,s_8,s_7,s_1,s_2,s_3,s_4,s_6,s_9,s_{10}\}$. As s_5 is ranked the most suspicious statement by Ochiai, three new test cases covering s_5 (t_6, t_7 and t_8) are generated and added into initial test cases in the first iteration. According to the coverage and test results of the eight test cases, Ochiai generates a new ranking list, $\{s_8,s_7,s_5,s_1,s_2,s_3,s_4,s_6,s_9,s_{10}\}$, in the first iteration. The top 20% of statements in ranking list of the initial running are $\{s_5,s_8\}$ while those of the first iteration are $\{s_8,s_7\}$. As $\{s_5,s_8\}$ is different from $\{s_8,s_7\}$, the iteration will continue. As s_8 is ranked the most suspicious statement in the first iteration, another three new test cases covering s_8 (t_9, t_{10} and t_{11}) are generated and added into all previous test cases in the second iteration. Based on the coverage and results of the eleven test cases, Ochiai produces a new ranking list, $\{s_8,s_7,s_1,s_2,s_3,s_4,s_6,s_5,s_9,s_{10}\}$, in the second iteration. The top 20% of statements in the ranking of the first iteration are $\{s_8,s_7\}$ and those of the second iteration are $\{s_8,s_7\}$. Because

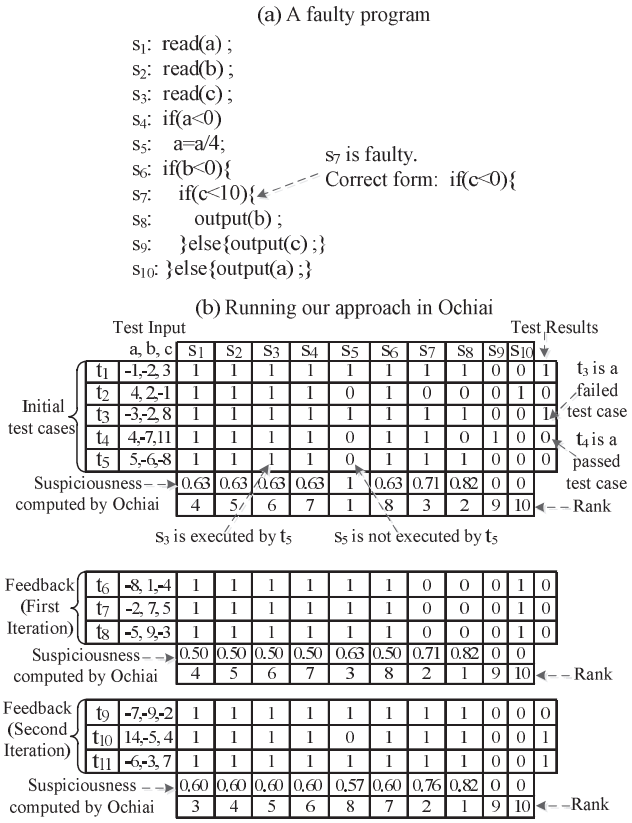

Fig. 5 Example illustrating the approach.

[†]A systematic study of other proper values of the threshold is part of our future work.

they are the same [†], our approach will terminate the iteration and output the ranking list of the second iteration as the localization result. Observe that the faulty statement s_7 is ranked third without feedback whereas s_7 is ranked second after applying our approach.

5. An Experimental Study

5.1 Experimental Setup

To evaluate the effectiveness of our approach, the experimental study chooses the *Siemens suite* and *Space* as our benchmarks because they are two widely used benchmarks in the field of fault localization [2], [4], [8], [9], [11], [12], [15]–[21], [25]. They cover a wide spectrum of faults with high quality, such as predicate faults, assignment faults, missing code, etc. They can be acquired from Software Information Repository [27]. The *Siemens suite* was originally developed at the Siemens Research Corporation and contains 7 programs and 132 faulty versions of these programs. The *Space* was first written by the European Space Agency and contains 38 faulty versions. In *Siemens suite*, each faulty version contains exactly one seeded fault. In *Space*, each faulty version includes exactly one realistic fault. Table 1 lists the programs, the number of faulty versions of each program, lines of statements, lines of executable statements, number of all test cases, as well as the functional descriptions of the corresponding program. The lines of executable statements are obtained by ignoring unexecutable source code such as macro definitions, function and variable declarations, blank lines, comments, and function prototypes.

There are test suites satisfying different test criteria in the *Siemens suite* and *Space*. The experiment selects the *universe* suite [††] whose test adequacy is the most powerful among all test suites of the *Siemens Suite* and *Space*. As the *universe* suite consists of a large number of test cases, the information should be fairly adequate for fault localization. If our approach obtains significant improvement, it can also fully demonstrate that test cases from testing are likely to offer information that is inadequate for conducting efficient fault localization. The *universe* suite contains all test cases in Table 1.

Although there are 170 versions in total, we were unable to adopt all of them. Because there were no failed test case in version 32 of *replace*, version 9 of *schedule2* and versions 1, 2, 34 of *Space*, we excluded the five versions. Moreover, our interests focus on executable statements, so the modifications of header files and definition/declaration errors were ignored. Hence versions 4 and 6 of *print_tokens*, version 12 of *replace*, versions 13, 14, 36, 38 of *tcas* and versions 6, 10, 19, 21 of *tot_info* were also discarded. In all, 3 faulty versions of *Space* and 13 faulty versions of *Siemens suite* were discarded by the experiment. Finally, 154 faulty versions were used for the experiment.

As SFL is a representative and promising technique widely used and studied in the fault localization community [2]–[5], [8]–[21], our approach is applied to three typical ranking metrics of SFL techniques that are Ochiai [9], Jaccard [10] and Tarantula [11] respectively. Our approach is implemented according to the implementation in Sect. 4.1. Additionally, our approach is compared with two SFL refinement approaches (referred as ISSFL [20] and TG [21]).

Table 1 Description of the *Siemens Suite* and *Space*.

Program	versions	LOC	Ex	Test	Description
print_tokens	7	563	203	4130	Lexical analyzer
print_tokens2	10	508	203	4115	Lexical analyzer
replace	32	563	289	5542	Pattern recognition
schedule	9	410	162	2650	Priority scheduler
schedule2	10	307	144	2710	Priority scheduler
Tcas	41	173	67	1608	Altitude separation
tot_info	23	406	136	1052	Information measure
Space	38	9564	6218	13585	ADL interpreter

ISSFL gives those unexecuted statements in all failed test cases the lowest suspiciousness, and the suspiciousness of the other statements is assigned by SFL. TG first sorts statements executed by the same number of failed test cases into a group. Then, the group with a larger number of failed test cases is ranked a higher position. Finally, the statements of each group are ranked in descending order of suspiciousness assigned by SFL.

5.2 Evaluation Metric

The performance of SFL is widely evaluated by the percentage of code that needs to be examined (or not examined) to find the fault [2]–[5], [8], [9], [11]–[15], [17]–[21]. This evaluation assumes that debugging engineers will examine all statements from top to bottom according to the ranking list given by the SFL until they encounter the faulty statement. Following this convention, we define the *fault-localization accuracy* (referred as *Acc*) as the percentage of executable statements to be examined before finding the actual faulty statement [21]. A lower value of *Acc* indicates better performance.

For a more detailed comparison, a metric named *performance improvement* (referred as *Imp*) is adopted by the experiment. *Imp* is the relative decrease in *Acc* after applying our approach [20] (see Eq. (6)).

$$Imp = (Acc_b - Acc_a)/Acc_b \tag{6}$$

In Eq. (6), Acc_b is the *Acc* of SFL before applying our approach. Acc_a is the *Acc* of SFL after applying our approach. A lower value of *Imp* shows better improvement that our approach obtains.

In addition, *relative effectiveness* (referred as *R*) is defined to compare the effectiveness of new test cases with that

[†]The comparison includes the ranking order of the statements. For example, $\{s_8,s_7\}$ is different from $\{s_7,s_8\}$ because the ranking orders of the statements in the two sets are different.

[††]In the *universe* suite, each executable statement and edge in the program or its control flow graph was exercised by at least 30 test cases.

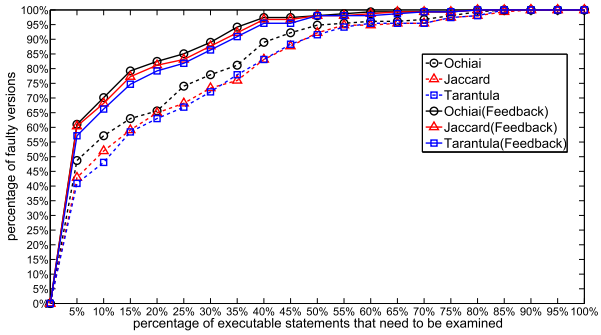

Fig. 6 *Acc* comparison between original SFL and our approach in all faulty versions.

of initial test cases as follows:

$$R = \frac{((1 - Acc_a) - (1 - Acc_b))/NewTestNum}{(1 - Acc_b)/InitialTestNum} = \frac{(Acc_b - Acc_a)/NewTestNum}{(1 - Acc_b)/InitialTestNum} \quad (7)$$

In Eq. (7), the meanings of Acc_b and Acc_a refer to Eq. (6). $NewTestNum$ denotes the number of new test cases that are added to initial test cases. $InitialTestNum$ is the number of initial test cases. $(Acc_b - Acc_a)/NewTestNum$ represents new test cases' average contribution to the performance of SFL, whereas $(1 - Acc_b)/InitialTestNum$ denotes initial test cases' average contribution to the performance of SFL. When $R > 1$, it indicates that the average effectiveness of new test cases is higher than that of initial test cases.

When using the metrics above, two types of faults should be specified how they are examined. One is the single fault spanning multiple statements. We assume that when examining any of these multiple statements, programmers can locate this type of fault. The other is the single fault related to missing code. The previous strategy [11], [17] only considers the statement preceding the missing code to be the faulty statement. Besides the preceding one, we would consider the statement succeeding the missing code because the two statements would attract programmers' attention to the missing code. We assume that developers can identify the missing code when inspecting any of the two statements.

5.3 Results and Analysis

Figure 6 presents *Acc* comparison between the original SFL and SFL with our approach in all faulty versions. The x-axis represents the percentage of executable statements to be examined. The y-axis denotes the percentage of faulty versions. A point in Fig. 6 represents when a percentage of executable statements is examined in each faulty version, the percentage of faulty versions has located their faults.

As shown in Fig. 6, the curves of Ochiai, Jaccard and Tarantula with our approach are always higher than they alone. That suggests the performance of all three ranking metrics of SFL is apparently improved by our approach.

Figure 7 illustrates average *Acc* comparison between original SFL and our approach in each type of faults. In

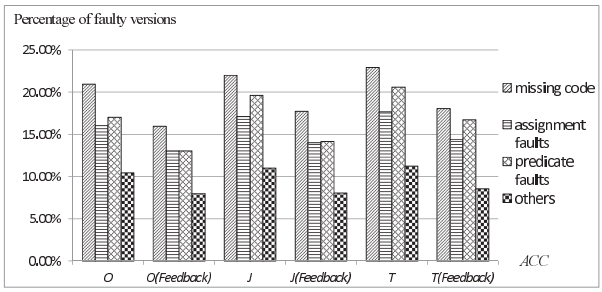

Fig. 7 Average *Acc* comparison between original SFL and our approach in each type of faults.

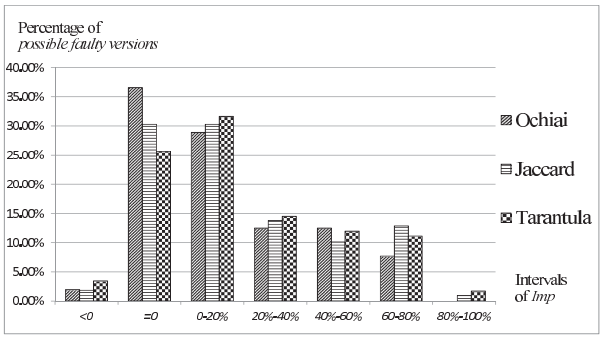

Fig. 8 Distribution of *Imp*.

Fig. 7, *O*, *J*, *T* and *(Feedback)* represent Ochiai, Jaccard, Tarantula and the corresponding approach using feedback respectively. This study classifies all faults into four types, namely missing code, assignment faults, predicate faults and others. The type of others includes additional code, return faults, multiple-positions faults, etc. As shown in Fig. 7, after applying our approach, the average percentage of executable statements to be examined significantly decreases in all three metrics of SFL as they locate each type of faults. This result indicates that our approach improves all three metrics of SFL in locating each type of faults.

Before applying the proposed approach, there are some faulty versions whose actual faulty statements are ranked first by SFL. These faulty versions are named *impossible faulty versions* and the other faulty versions are named *possible faulty versions*. Due to the situation where the improvement is impossible, we excluded *impossible faulty versions* from the following analysis of *Imp*. Please note that the faults of *impossible faulty versions* are still ranked first after applying our approach.

For a more detailed comparison, Fig. 8 shows the distribution of *Imp* in all possible faulty versions. The x-axis represents the intervals of *Imp*. The y-axis denotes the percentage of *possible faulty versions* that belongs to a specific interval of *Imp*. There are seven intervals for *Imp*: *<0*, *=0*, *0–20%*, *20%–40%*, *40%–60%*, *60–80%* and *80%–100%*. The interval of *<0* indicates that our approach decreases the performance of SFL. The interval of *=0* represents that no improvement in SFL is obtained after applying our approach. The other intervals suggest that our approach improves SFL.

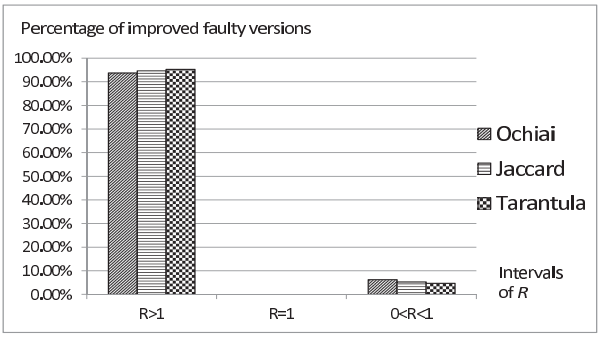

Fig. 9 Distribution of R for improved faulty versions.

The interval with a higher value of *Imp* represents the better improvement.

As shown in Fig. 8, 61.5% of *possible faulty versions* are improved by our approach with Ochiai, 67.9% with Jaccard and 70.9% with Tarantula. An average of 30.2% of *possible faulty versions* is improved by our approach at the interval of *0–20%*, 13.6% at the interval of *20%–40%*, 11.5% at the interval of *40%–60%* and 10.5% at the interval of *60%–80%*. Few possible faulty versions obtain an *Imp* at the interval of *80%–100%*.

There are two reasons for the result as shown in Fig. 8. One is that the three representative metrics of SFL have already output fine results for most faulty versions before applying our approach, which provides limited room for improving SFL. The other is that whatever a fault localization approach is, it is genuinely difficult to obtain an *Imp* at the interval of *80%–100%*. Even so, our approach still obtains the significant improvement over SFL.

We observe that our approach decreases the performance of SFL in several faulty versions. We find that these statements are always highly associated with faulty outputs according to data/control dependencies. Based on the dependencies, we conjecture that these statements may have a higher *GFP* than the faulty statement. If this conjecture is established, it suggests that former ranking list is far away from the theoretical reasonable result of FL Rule mentioned in Sect. 3.1, and our approach just makes the localization result become more reasonable. Furthermore, our approach only leads to a minor performance decrease in these faulty versions. Thus the performance decrease can be ignored.

Figure 9 presents the distribution of *relative effectiveness* for faulty versions whose performance of SFL has been improved by our approach. The x-axis represents the intervals of R. There are three intervals of *relative effectiveness*, as $R > 1$, $R = 1$, $0 < R < 1$. The y-axis denotes the percentage of faulty versions whose performance of SFL has been improved by our approach. As shown in Fig. 9, for each type of SFL, more than 93% of improved faulty versions are at the interval of $R > 1$. This suggests that once SFL is improved by our approach, the average effectiveness of new test cases is always higher than that of initial test cases.

Table 2 lists the programs, the average number of new test cases, the percentage of the average number of new test cases in the number of initial test cases. Table 2 shows that the average number of new test cases only accounts for a minor percentage of the number of initial test cases. It also suggests new test cases have high effectiveness.

Table 2 Description of new test cases.

Program	Average new test cases	Average new test cases in initial test cases
print_tokens	98	2.37%
print_tokens2	84	2.04%
replace	109	1.97%
schedule	63	2.38%
schedule2	80	2.95%
tcas	47	2.92%
tot_info	34	3.23%
Space	252	1.85%

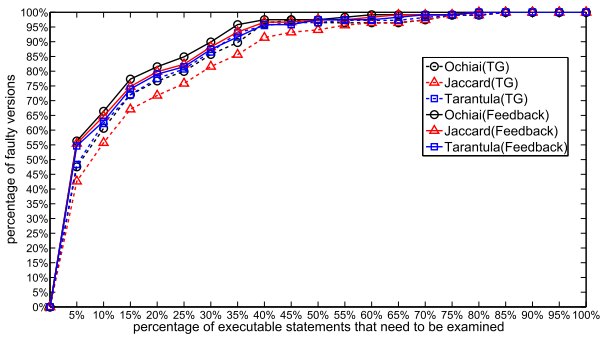

Fig. 10 *Acc* comparison between TG and our approach.

Because ISSFL is only applicable to the common metrics of SFL, it cannot be applied to the three fine metrics of SFL adopted by this study. However, *Acc* comparison between two SFL refinement approaches can be made when the two approaches are applied to the same metric of SFL. Hence, this study cannot conduct the *Acc* comparison between ISSFL and our approach. In addition, there is an assumption when using *Imp* to compare ISSFL with our approach. The assumption is that if the *Imp* in the better metric of SFL is higher than that in the common metric of SFL, the approach applied to the better metric of SFL should outperform the approach applied to the common metric of SFL. As the original data from the TG's published work [21] is not sufficient to conduct the analysis of *Imp*, we implement the TG according to its algorithm described in [21] to compare TG with our approach in *Imp*. Following the same experimental condition in ISSFL [20] and TG [21], our approach will be compared with the two SFL refinement approaches on the *Siemens suite*.

Figure 10 denotes the *Acc* comparison between TG and our approach. As shown in Fig. 10, the curves of TG are always slightly beneath their corresponding curves of our approach. This indicates that our approach performs a bit better than TG.

Figure 11 presents the average *Imp* comparison in ISSFL, TG and our approach. As shown in Fig. 11, the average *Imp* of our approach is higher than ISSFL in all programs except the program of *schedule*. In the four out of

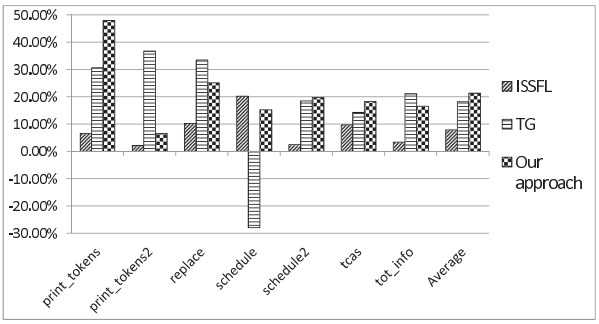

Fig. 11 Average *Imp* comparison in ISSFL, TG and our approach.

seven programs, the average *Imp* of our approach is higher than TG. On average, the *Imp* of our approach is also higher than ISSFL and TG. Therefore, our approach significantly outperforms ISSFL and obtains the minor improvement over TG.

From all results discussed above, we can conclude that: 1) The proposed approach is effective to improve the performance of SFL. 2) Merely relying on test cases from testing are likely to offer information that is inadequate for conducting efficient fault localization.

5.4 Threats to Validity

First and foremost, the proposed approach assumes that fault localization methods have explicitly or inexplicitly applied FL Rule. As SFL has applied FL Rule, our approach obtains the significant improvement in SFL. If a fault localization method does not apply FL Rule, the result may be misleading.

Second, the distribution of *GFP* in all program statements may affect the performance of our approach. For instance, if there are many statements in *affected set* and their *GFPs* are fairly close to that of the faulty statement. Due to high degree of similarity, it is genuinely difficult for our approach to make the *FP* of these statements lower than that of the faulty statement. Under this condition, the performance of our approach may become weak.

Next, the improvement in SFL depends on the effectiveness of initial test cases. To demonstrate test cases from testing are likely to offer inadequate information for fault localization, the experiment chooses the *universe* suite that is the most powerful test suite in the *Siemens suite* and *Space*. However, maybe some other powerful test adequacy criteria can make test cases provide adequate information for fault localization. As a result, our approach may become misleading. It is vital to conduct a further study on the effectiveness of our approach in different test adequacy criteria.

Another threat is the subject programs used by the experiment. The *Siemens suite* and *Space* are two standard benchmarks widely used in the field of fault localization. The faults of the two benchmarks cover a wide spectrum of realistic faults with high quality. Thus the experimental results should be reliable. However, we studied only single-fault versions. Apparently, the results obtained may not apply to all programs. For instance, a program, in reality, usually ships with multiple faults rather than a single fault as used in our experiment. In addition, new test cases may trigger and cause new failures. The recent research [4] has found that multiple faults pose a negligible effect on the effectiveness of the fault localization, and it can be guaranteed that even in the presence of many faults, at least one fault is found by SFL with high effectiveness despite the effect of fault-localization interference. These findings increase our confidence in the effectiveness of our approach for locating multiple faults. Nevertheless, there are still many unknown and complicated factors in the realistic debugging, which may lead our approach to be misleading. Thus, it is worthwhile to use more subject programs, such as multiple-faults programs, to further validate the effectiveness of our approach for fault localization.

6. Related Work

SFL has attracted considerable attention in recent years and motivates plenty of research in the field of fault localization. There are many ranking metrics of SFL, such as Optimal [8], ISSFL [20], TG [21] and the three representative metrics of SFL adopted by this study [9]–[11]. To strengthen the relationship among the elements of a program entity, some new complex coverage types of program entities using dependences or flow are proposed for SFL, such as mixed coverage [12], information flow coverage [13], and control flow edge coverage [14]. To support locating multiple faults in parallel, Jones et al. [15] adopt a clustering technique to divide failed test cases into different clusters and each cluster represents one fault. Abreu et al. [16] propose a spectrum-based multiple fault localization approach called Zoltar-M that integrates SFL with model-based diagnosis. To address the coincidental correctness problem in coverage-based fault localization approaches, Wang et al. [17] prescribe patterns for different types of faults and modify the coverage vectors to locate faults with the feature of coincidental correctness. To solve test oracle problem, Abreu et al. [18] apply invariants to SFL and use error detection to judge failures, and Xie et al. [19] apply metamorphic testing to SFL to perform SFL without test oracle. It can be seen that current research on SFL usually ignores the interaction between software debugging engineers and fault localization. However, our approach simulates the interaction to address the problem of insufficient interactions. In addition, besides the three representative types of SFL in this study, these approaches can be also adopted by our approach, which is part of our current research.

Shapiro [22] proposes a diagnosis approach to realize the interaction between debugging engineers and fault localization by asking debugging engineers questions and utilizing answers to locate faults. Fritzson et al. [23] adopt the category partition testing technique to refine the questions for debugging engineers, and use slicing technique to further narrow down engineers' searching scope. To support common users in debugging, three interactive fault local-

ization approaches are presented in [24]. In addition, two factors that influence interactive performance are also analyzed in [24]. Hao et al. [25] use breakpoints to interact with debugging engineers and can somewhat rectify mistakes made by engineers. The above interactive fault localization approaches still need intensive workload of debugging engineers in analysis and deduction which incurs a lot of overhead. Unlike these interactive fault localization approaches, our approach simulates the pattern of debugging engineers as they analyze and deduce faults in the interaction to improve fault localization performance and further reduce debugging engineers' workload.

7. Conclusion

Following the interaction seen between software debugging engineers and fault localization, this paper proposes an effective approach using feedback to simulate that interaction. In essence, our approach attempts to simulate the approach of debugging engineers as they apply their knowledge and experience in this interaction. Although it is difficult to imitate debugging engineers' brainpower, we still see that the proper simulation of this interaction has great potential for improving fault localization performance. Additionally, it also indicates that test cases from testing are likely to offer information that is inadequate for conducting efficient fault localization. Using feedback is a practical approach to address this problem.

In future work, we plan to evaluate the effectiveness of our approach across a much broader spectrum of programs, especially in the context of multiple faults. We also wish to consider further optimizations to our approach that will lead to more improvements in terms of fault localization effectiveness, such as other proper values of the threshold. Additionally, the applicability of our approach to other ranking metrics of SFL will be also studied.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant No.91118007, 90818024 and 61133001, the National High Technology Research and Development Program of China (863 program) under Grant No.2011AA010106 and Program for New Century Excellent Talents in University.

References

[1] A. Zeller, Why programs fail: A guide to systematic debugging, Morgan Kaufmann, 2005.

[2] J.A. Jones and M.J. Harrold, "Empirical evaluation of tarantula automatic fault-localization technique," Proc. 20th International Conference on Automated software engineering, pp.273–282, Long Beach, CA, USA, 2005.

[3] S. Ali, J.H. Andrews, T. Dhandapani, and W. Wang, "Evaluating the accuracy of fault localization techniques," Proc. 2009 IEEE/ACM International Conference on Automated Software Engineering, pp.76–87, Auckland, New Zealand, 2009.

[4] N. DiGiuseppe and J. Jones, "On the influence of multiple faults on coverage-based fault localization," Proc. 2011 International Symposium on Software Testing and Analysis, Toronto, ON, Canada, 2011.

[5] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," Proc. 28th International Conference on Software Engineering, pp.82–91, Shanghai, China, 2006.

[6] J. Wang, X.D. Ma, W. Dong, H.F. Xu, and W.W. Liu, "Demand-driven memory leak detection based on flow-and context-sensitive pointer analysis," J. Computer Science and Technology, vol.24, no.2, pp.347–356, 2009.

[7] T. SHIMOMURA, "Critical slice-based fault localization for any type of error," IEICE Trans. Inf. & Syst., vol.E76-D, no.6, pp.656–667, June 1993.

[8] L. Naish, H.J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," ACM Trans. Softw. Eng. Methodol., vol.20, no.3, pp.1–32, 2011.

[9] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "On accuracy of spectrum-based fault localization," Proc. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, pp.89–98, Windsor, UK, 2007.

[10] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," Proc. 32nd International Conference on Dependable Systems and Networks, pp.595–604, Maryland, USA, 2002.

[11] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," Proc. 24th International Conference on Software Engineering, pp.467–477, Orlando, Florida, 2002.

[12] R. Santelices and J.A. Jones, "Lightweight fault-localization using multiple coverage types," Proc. 31st International Conference on Software Engineering, pp.56–66, Vancouver, Canada, 2009.

[13] W. Masri, "Fault localization based on information flow coverage," Software Testing, Verification and Reliability, vol.20, no.2, pp.121–147, 2010.

[14] Z. Zhang, W. Chan, T. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," Proc. 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.43–52, Amsterdam, The Netherlands, 2009.

[15] J.A. Jones, J.F. Bowring, and M.J. Harrold, "Debugging in parallel," Proc. 2007 International Symposium on Software Testing and Analysis, pp.16–26, London, United Kingdom, 2007.

[16] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "Simultaneous debugging of software faults," J. Systems and Software, vol.84, no.4, pp.573–586, 2010.

[17] X. Wang, S. Cheung, W. Chan, and Z. Zhang, "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization," Proc. 31st International Conference on Software Engineering, pp.45–55, Vancouver, British Columbia, Canada, 2009.

[18] R. Abreu, A. González, P. Zoeteweij, and A.J.C. van Gemund, "Automatic software fault localization using generic program invariants," Proc. 2008 ACM Symposium on Applied Computing, Fortaleza, pp.712–717, Ceara, Brazil, 2008.

[19] X. Xie, W.E. Wong, T.Y. Chen, and B. Xu, "Spectrum-based fault localization: Testing oracles are no longer mandatory," Proc. 11th International Conference On Quality Software, pp.1–10, Madrid, Spain, 2011.

[20] X. Xie, T.Y. Chen, and B. Xu, "Isolating suspiciousness from spectrum-based fault localization techniques," Proc. 10th International Conference on Quality Software, pp.385–392, Zhangjiajie, China, 2010.

[21] V. Debroy, W.E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve effectiveness of fault localization techniques," Proc. 10th International Conference on Quality Software, pp.13–22, Zhangjiajie, China, 2010.

[22] E.Y. Shapiro, "Algorithmic program diagnosis," Proc. 9th ACM SIGPLAN SIGACT symposium on Principles of programming lan-

guages, pp.299–308, Albuquerque, New Mexico, USA, 1982.
[23] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy, "Generalized algorithmic debugging and testing," ACM Letters on Programming Languages and Systems, vol.1, no.4, pp.303–322, 1992.
[24] J.R. Ruthruff, M. Burnett, and G. Rothermel, "Interactive fault localization techniques in a spreadsheet environment," IEEE Trans. Softw. Eng., vol.32, no.4, pp.213–239, 2006.
[25] D. Hao, L. Zhang, T. Xie, H. Mei, and J.S. Sun, "Interactive fault localization using test information," J. Computer Science and Technology, vol.24, no.5, pp.962–974, 2009.
[26] J. Edvardsson, "A survey on automatic test data generation," Proc. 2nd Conference on Computer Science and Engineering, pp.21–28, Linkoping, Sweden, 1999.
[27] SIR, http://sir.unl.edu.

Yan Lei is currently a Ph.D. candidate in computer science at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software debugging.

Xiaoguang Mao is currently a full professor at School of Computer, National University of Defense Technology, 410073, Changsha, China. He received his Ph.D. degree in computer science from National University of Defense Technology in 1997. His research interests include high confidence software, software development methodology, software assurance, software service engineering, etc.

Ziying Dai is currently a Ph.D. candidate in computer science at School of Computer, National University of Defense Technology, 410073, Changsha, China. His research interests include software dynamic evolution.

Dengping Wei is currently a lecturer at School of Computer, National University of Defense Technology, 410073, Changsha, China. She received her Ph.D. degree in computer science from National University of Defense Technology in 2011. Her research interests include Semantic Web, Web service, information retrieval and program analysis.