Improving the Accuracy of Spectrum-Based Fault Localization Using Multiple Rules

Rongcun WANG†††(a), Shujuan JIANG(b), Kun ZHANG(c), Nonmembers, and Qiao YU†††(d), Member

SUMMARY Software fault localization, as one of the essential activities in program debugging, aids to software developers to identify the locations of faults in a program, thus reducing the cost of program debugging. Spectrum-based fault localization (SBFL), as one of the representative localization techniques, has been intensively studied. The localization technique calculates the probability of each program entity that is faulty by a certain suspiciousness formula. The accuracy of SBFL is not always as satisfactory as expected because it neglects the contextual information of statement executions. Therefore, we proposed 5 rules, i.e., random, the maximum coverage, the minimum coverage, the maximum distance, and the minimum distance, to improve the accuracy of SBFL for further. The 5 rules can effectively use the contextual information of statement executions. Moreover, they can be implemented on the traditional SBFL techniques using suspiciousness formulas with little effort. We empirically evaluated the impacts of the rules on 17 suspiciousness formulas. The results show that all 5 rules can significantly improve the ranking of faulty statements. Particularly, for the faults difficult to locate, the improvement is more remarkable. Generally, the rules can effectively reduce the number of statements examined by an average of more than 19%. Compared with other rules, the minimum coverage rule generates better results. This indicates that the application of the test case having the minimum coverage capability for fault localization is more effective.

key words: fault localization, spectrum, rule, accuracy

1. Introduction

Software testing and debugging are very time-consuming and expensive activities in the whole software development lifecycle. The two activities account for 50% to 80% of the total development and maintenance effort [1]. Fault localization is one of the most essential activities. Manual fault localization can be time-consuming and prohibitively expensive, as well as tedious and error-prone [2]. Therefore, many fault localization techniques [3]–[7] have been proposed to achieve the semi-automation or automation of software debugging with the aims of decreasing the cost of software debugging, as well as improving the software quality [8].

Spectrum-based fault localization, as a promising approach, has been intensively studied in software engineering community. In general, the technique makes use of various program spectra information referring to any granularity of program entities [9] and the running result with respect to each test case collected from software testing to locate the suspicious code in program. After acquiring the necessary information, this technique calculates the suspiciousness value of each program entity by a certain suspiciousness formula. The program entities are ranked in descending order according to their suspiciousness values. Developer begins to examine the code according to the ranking list. The examination of suspicious code stops as long as one faulty program entity is identified since a program bug may span multiple lines of code [10]. In other words, if the localization technique can reduce the number of examined statements for the first fault, it is very meaningful although the program contains more than one fault.

SBFL only utilizes the execution result and the relationship between test cases and the program entities, while neglects the logical relationship between program entities, i.e., the contextual information of program entity executions. In fact, the contextual information is very useful to spectrum-based fault localization [11]. Moreover, SBFL treats all failed test cases equally. In other words, the contributions of all failed test cases to fault localization are not differentiated. Various suspiciousness formulas have been proposed for fault localization. None of them can provide a completely satisfactory solution.

Specifically, for large scale software, the complication of the logical relationship between codes further increases the difficulty of fault localization. Furthermore, the faulty program entities are not always ranked at the top of the ranking list generated by SBFL. In this context, developers have to examine more program entities unrelated to the faults before localizing the faulty program entities. This means that they inevitably pay more efforts for fault localization. If the program entities unrelated to the faults are completely or partially excluded, the rankings of the faulty program entities can be improved. As a result, developers only need to examine a small number of program entities before the fault is located. Unfortunately, the program entities unrelated to the faults cannot be known in advance of localizing the faults.

According to the PIE model [12], a fault can be de-
ected when the following three conditions are met: First, the location in the program that contains the fault must be reached, i.e., the faulty code can be executed; Second, the state of the program must be incorrect; Third, the infected state must propagate to cause the output of the program to be incorrect. To sum up, the program entity containing a fault can be executed and the testing result is failed. As aforementioned, it is difficult to identify the program entities unrelated to the faults. On the contrary, we can easily identify the program entities potentially related to the faults. Before localizing the faulty program entities, we call the entities executed by the failed test cases to be potentially faulty program entities. Obviously, the program entities executed by the failed test cases have higher possibility to be faulty [13].

The locations containing the faults and the number of the faults, as well as the corresponding relationship between failed test cases and the locations are unknown. Nevertheless, the running result and the program spectrum information with respect to each test case is known. Moreover, a failed test case can detect one of all faults in the program under test at least even if we do not know that which fault can be detected by it. We attempt to make use of the program spectra of a failed test case to identify the set of the potentially faulty program entities, so as to reduce the impact of the program entities unrelated to the faults on SBFL as far as possible. Therefore, we proposed five rules, i.e., random, the maximum coverage, the minimum coverage, the maximum distance, and the minimum distance to identify the failed test case, respectively. Program spectra of the test case identified by one of the rules is essentially a whole execution slice, which can effectively preserve the contextual information of program entity execution.

We conducted and designed empirical studies on the Siemens suite to evaluate the impacts of the five rules on 17 suspiciousness formulas. The results show that all five rules can significantly improve the accuracy of 17 SBFL techniques. In general, the rules can reduce the number of statements examined by an average of more than 19%. The minimum coverage rule generates better results than other four rules. More importantly, the rules we proposed can be implemented on all SBFL techniques using different suspiciousness formulas with little effort.

The remainder of this paper is organized as follows. Related work and background are briefly introduced in Sect. 2 and Sect. 3, respectively. Section 4 describes the rules we proposed. Experimental design and results analysis are presented in Sect. 5. The threats to validity are discussed followed by the conclusion and future work in Sect. 6 and Sect. 7, respectively.

2. Related Work

Many fault localization techniques including slice-based, program spectrum-based, model-based, machine-based techniques, etc., have been proposed [10]. Since the rules are used to improve the accuracy of spectrum-based fault localization, we only discussed program spectrum-based fault localization techniques.

To address the year 2000 problem, the concept of program spectra was first introduced [14]. Harrold et al. [9] empirically evaluated various types of program spectra e.g., branch spectra, data dependence spectra, and execution trace spectra. The results confirm that faults correlate with spectra differences between a faulty and a correct run. This means the comparison of the differences between program spectra is usefulness for tracking program behavior. In other words, software diagnosis can benefit from the analysis of the differences between program spectra. Early studies [15], [16] only used failed test cases for SBFL though the method has been known to be ineffective [5], [17].

Later studies used failed test case as well as successful test cases to get better results. Reniers and Reiss [18] proposed a fault localization method based on the similarity of two program spectra, i.e., the spectra of a single failed test and a single successful test that is most similar to the failed test. Jones et al. [3] developed a visual system, called Tarantula, to assist fault localization. The Tarantula system uses the coverage information of statements and execution results from multiple failed and successful runs to calculate the ranking of each statement. The ranking can be indicated by a spectrum of colours. Furthermore, the extendibility of the Tarantula is very well, i.e., the system could easily be modified to replace the Tarantula metric with another metric. Subsequently, Jones et al. [17] evaluated the fault localization ability of Tarantula in a comparison with the techniques in [18].

AMPLE [19] is an Eclipse plug-in for locating buggy Java classes. The likelihood of each class being buggy is computed based on the difference in method call sequences between a single failing run and multiple passing runs. Abreu et al. [4] first introduced the Ochiai metric for software diagnosis and empirically compared it with Jaccard, AMPLE and Tarantula techniques using the Siemens test suite. Wong et al. [20] proposed three suspiciousness formulas for fault localization using code coverage.

With more and more suspiciousness formulas proposed, some studies began to empirically compare their performance [21]. Nevertheless, no technique is always superior to others under each scenario, i.e., the optimum suspiciousness formula does not exist [22]. In view of many limitations of the experimental study, some researchers have attempted to evaluate the performance of suspiciousness formulas from a theoretical perspective in recent studies. Lee et al. [23] have proved that the Tarantula formula is equivalent with q<sub>e</sub>, i.e., they always generate identical suspicious ranking list. Naish et al. [24] proposed a model-based approach to evaluate 33 suspiciousness formulas. They have proved several suspiciousness formulas are equivalent for ranking purposes using the same definition of equivalence as Lee et al. [23]. Similarly, Xie et al. [8] theoretically analyzed 30 suspiciousness formulas by a framework they developed. Different from the equivalence in [24], the equivalence defined in [8] is more looser.
Table 1: An example for SBFL using the Tarantula metric

| Line NO. | int foo(int x,int y,int z) | Test case | suspiciousness | Ranking |
|----------|-----------------------------|-----------|----------------|---------|
| 1        | int w = 1;                  | t1        | 1,1,1          | S       |
| 2        | if(x < 0) {}                | t2        | 0,1,1          | S       |
| 3        | w* = x; /bug, w* = -x      | t3        | -1,1,1         | F       |
| 4        | if(x < 0) {}                | t4        | -1,0,1         | F       |
| 5        | w* = -y;                    | t5        | -1,-1,1        | F       |
| 6        | if(z <= 0) {}               | t6        | -2,-1,-2       | F       |
| 7        | w* = z;                     | t7        | -1,-1,0        | F       |
| 8        | else if (z > 0) {}          |           | 0              | S       |
| 9        | w* = z;                     |           | 0              | S       |
| 10       | else { w = z; }             |           | 0              | S       |
| 11       | else if (y > 0) {}          |           | 0.57           | 7       |
| 12       | w* = y                      |           | 0              | 1       |
| 13       | else { w = y; }             |           | 0              | 11      |
| 14       | else if (x > 0) {}          |           | 0              | 11      |
| 15       | w* = x;                     |           | 0              | 11      |
| 16       | else { w = x; }             |           | 0              | 11      |
| 17       | printf("%d \n", w);       |           | 0.5            | 8       |

Success/Fail Status | S | S | F | S | F | F | S

3. Background

3.1 Spectrum-Based Fault Localization

A program spectrum describes the runtime profile information of a program from certain perspectives, such as branches, statements, basic blocks, or loop-free intraprocedural paths. It reflects the dynamic behavior of program. SBFL utilizes program spectrum and testing results collected during software testing to locate the faults. Therefore, the approach is a dynamic fault localization approach. The runtime profile information could be the number of times that the program entity has been covered, the binary coverage status, or the program status before and after executing the program entity. The testing result represents whether a test case is successful or failed. To date, the binary coverage status at the statement coverage level as an effective execution slice has attracted most attentions [8], [20], [24]. Therefore, we also adopted statement spectra with binary coverage status.

Given a program \( P = (s_1, s_2, \ldots, s_n) \) with \( n \) statements and a test suite \( T = (t_1, t_2, \ldots, t_m) \) containing \( m \) test cases. SBFL requires to collect the program spectrum information represented as coverage matrix \( M \), as well as the testing results, denoted as \( R = (r_1, r_2, \ldots, r_m) \). The element in the \( i \)th row and \( j \)th column of \( M \) represents the relationship between statement \( s_i \) and test case \( t_j \). If statement \( s_i \) is covered by test case \( t_j \), the element is equal to 1, otherwise 0. The element \( r_j \) of \( R \) denotes the testing result of test case \( t_j \) on \( P \). For each statement \( s_i \), we can define a tuple \( MR_i = (np(s_i), nf(s_i), ep(s_i), ef(s_i)) \) to represent the number of times that \( s_i \) has been executed or not executed, \( np(s_i) \) and \( nf(s_i) \) denote the number of successful or failed test cases in \( T \) that do not execute \( s_i \), respectively; \( ep(s_i) \) and \( ef(s_i) \) represent the number of successful or failed test cases that execute \( s_i \), respectively. The sum of these four elements in \( MR \) is always equal to the size of test suite \( T \).

Table 1 shows an example for SBFL, where \( S \) and \( F \) represent that the testing is successful or failed, respectively. Program \( P \) contains seventeen statements \( \{s_1, s_2, \ldots, s_{17}\} \), and test suite \( T \) has seven test cases \( \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\} \). In particular, \( t_3, t_5 \) and \( t_6 \) are failed and the remaining test cases are successful. The binary coverage information for each statement with respect to every test case is shown in Table 1, where the symbol \( \bullet \) denotes a statement can be covered. For example, \( MR_3 \) corresponding to statement \( s_3 \) is equal to \((2, 0, 2, 3)\). \( ep(s_3) = 2 \) means that \( s_3 \) is executed by two successful test cases.

SBFL calculates the suspiciousness value of each statement by a certain suspiciousness formula after completing the transformation from the spectrum information and the testing results to the four tuples. The suspiciousness value of each statement indicates the possibility to be faulty. For instance, the suspiciousness formula Tarantula is defined as follows:

\[
\text{suspiciousness}(s_i) = \frac{ef(s_i)}{ef(s_i) + ep(s_i)}. 
\]

All statements are listed in descending order according to their suspiciousness values after assigning the suspiciousness values to the statements. The last two columns in Table 1 present the suspiciousness values calculated by Tarantula and the rankings, respectively. A statement with higher suspiciousness value is examined with higher priority. Therefore, developers start debugging program according to the rankings of the statements after generating the ranking list.

3.2 Suspiciousness Formula

One of SBFL’s popular research areas is to design suspi-
Table 2 Suspiciousness formulas used in this paper

| Name               | Mathematical formula |
|--------------------|----------------------|
| AMPL2              | $\frac{ef}{n_{ep}} - \frac{ep}{n_{ep}}$ |
| Arithmetic Mean    | $2\frac{ef}{n_{ep}+n_{np}} - \frac{ep}{n_{ep}} + \frac{np}{n_{np}}$ |
| Cohen              | $\frac{4ef}{n_{ep}+n_{np}+(ef+ep)^2} - \frac{(ef)^2}{(ef+n_{ep}+n_{np})^2}$ |
| Fleiss             | $\frac{ef}{n_{ep}+n_{np}}$ |
| Jaccard            | $\frac{ef}{n_{ep}}$ |
| Kulczynski1        | $\frac{ef}{n_{ep}+n_{np}}$ |
| Kulczynski2        | $\frac{ef}{n_{ep}+n_{np}} + \frac{ep}{n_{ep}}$ |
| M1                 | $\frac{ef}{n_{ep}+n_{np}}$ |
| M2                 | $\frac{ef}{n_{ep}+n_{np}}$ |
| Naish              | $\begin{cases} -1 & \text{if } ef < F \\ F - ep & \text{if } ef = F \end{cases}$ |
| Ochiai             | $\frac{ef}{n_{ep}+n_{np}}$ |
| Scott              | $\frac{4ef-4n_{np}(ef+ep^2)}{(ef+n_{ep}+n_{np})^2}$ |
| Tarantula          | $\frac{ef}{n_{ep}} + \frac{ep}{n_{ep}}$ |
| Wong1              | $\frac{ef}{n_{ep}+n_{np}}$ |
| Wong2              | $\frac{ef}{n_{ep}+n_{np}}$ |
| Wong3              | $\begin{cases} ef - ep & \text{if } ep \leq 2 \\ ef - 2 - 0.1(ep - 2) & \text{if } 2 < ep \leq 10 \\ 2.8 - 0.001(ep - 10) & \text{if } ep > 10 \end{cases}$ |

Since the whole test suite contains more than one failed test case, determining how to select a failed test case is very key. An alternative method is to randomly select a failed test case from all failed test cases. The rule is called to be random rule. Additionally, we proposed two types of rules: coverage-based rule, i.e., the maximum coverage and the minimum coverage, and distance-based rule, i.e., the maximum distance and the minimum distance, to narrow the search space for faulty statements that made the execution to fail. In fact, the rules provide different ways to select a failed test case.

4.1 Coverage-Based Rule

Coverage-based rule uses coverage information to select a representative test case. The rule only considers the coverage capability of failed test cases. To address the challenge, we defined the upper and lower bounds of the number of the really faulty statements. The upper bound of the number of the really faulty statements is equal to the number of the failed test cases having the maximum coverage capability while the lower bound is equal to the number of the statements executed by the failed test cases having the minimal coverage capability in theory. If the test case having the minimal coverage capability is selected, the rule is called the minimal coverage rule. Likewise, if the test case having the maximum coverage capability is selected, the rule is called the maximum coverage rule. Since more than one test case may have the same maximum or minimum coverage capability, we can randomly select one.

4.2 Distance-Based Rule

Distance-based rule utilizes distance information between pair-wise test cases to select a representative test case. Different from coverage-based rule, distance-based rule considers the difference between failed test cases. The $i$th column in coverage matrix $M$ denotes the statement coverage information of test case $t_i$ representing as a vector $V_i = \langle v_{i1}, v_{i2}, \ldots, v_{in} \rangle$. Likewise, the statement coverage vector of test case $t_i$ is denoted as $V_j = \langle v_{j1}, v_{j2}, \ldots, v_{jn} \rangle$. Since each bit in a coverage vector is 0 or 1, the distance between test case $t_i$ and $t_j$ is defined as follows:

$$d(t_i, t_j) = \sum_{k=1}^{n} (v_{ik} \oplus v_{jk}).$$

The idea of distance has been used to address the issues of test case generation [25] and prioritization [26]–[28]. The more diverse test cases are, i.e., the test cases have the maximum distances to other test cases in the same test suite, the more faults they can detect [25], [26]. Therefore, we employed the idea to select a failed test case having the maximum distance to other failed test cases, i.e., the average of the distance sum between the test case and other failed test cases is the maximum. We call this the maximum distance...
Cluster analysis has been proposed and applied to select a representative subset of test cases \[29\]. Test cases in the same cluster have the similar fault detection capability. The centroid of a cluster represents the cluster itself. In general, the test cases having the minimum average distance to other test cases within the same cluster is viewed as the centroid. With that in mind, we also selected the failed test case having the minimum average distance to other failed test cases, i.e., the average of the distance sum between the test case and other failed test cases is the minimum. We call this the minimal distance rule. Like coverage-based rules, distance-based rules randomly select one if more than one test case has the same average distance to other failed test cases.

After implementing SBFL, a failed test case can be selected by one of the rules. The statements executed by the selected test case is the potentially faulty statements. The statements in the final ranking list are those statements belonging to the intersection of the set of the statements in the ranking list generated by SBFL and the set of the potentially faulty statements. Thus, the statements not belonging to the set of the potentially faulty statements, as well as at the top of the ranking list generated by SBFL are excluded. The implementation of the rules on SBFL do not require to collect additional information as the required information has been collected in the process of implementing SBFL. Therefore, the rules can be widely applied in SBFL techniques using the suspiciousness formulas with little effort.

Assume that the program under test contains more than one fault. The selected test case only detects a fault. The statement containing the fault detected by the selected test case necessarily belongs to the set of the potentially faulty statements according to the PIE model. On the contrary, the remaining faulty statements do not necessarily belong to the set. In other words, even if the selected test case cannot detect all faults located in the program, it can help developers find a good starting location to initiate the bug-fixing process, rather than to provide the completely faulty statements with respect to each bug \[10\].

Having seen Table 1, the suspiciousness list generated by Tarantula is \(s_1 \rightarrow s_{12} \rightarrow s_5 \rightarrow s_6 \rightarrow s_3 \rightarrow \cdots \rightarrow s_{16}\). Tarantula needs to examine five statements to detect the fault in \(s_3\). The failing test cases are \(t_3, t_5\) and \(t_6\). According to the definition of the minimum coverage rule, \(t_3\) is selected with higher priority because the number of the statements covered by \(t_3\) is less than \(t_5\) and \(t_6\). The set of the statements covered by \(t_3\), i.e., the set of the potentially faulty statements is \(S_1 : \{s_1, s_2, s_3, s_4, s_{11}, s_{12}, s_{17}\}\). The final ranking list is \(s_{12} \rightarrow s_3 \rightarrow s_4 \rightarrow s_{11} \rightarrow s_1 \rightarrow s_2 \rightarrow s_{17}\) after using the minimum coverage rule. \(s_3\) is ranked second. This means that SBFL based on Tarantula only needs to examine two statements to detect the fault in \(s_3\) after using the minimum coverage rule. Similarly, \(t_5\) or \(t_6\) can be selected based on the maximum coverage rule as the coverage scores of them are equal to each other and the largest. The set of the potentially faulty statements identified by \(t_5\) or \(t_6\) is \(S_2 : \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_{17}\}\). We exclude those statements not belonging to \(S_2\) from the ranking list generated by Tarantula. The final ranking is \(s_7 \rightarrow s_5 \rightarrow s_6 \rightarrow s_3 \rightarrow s_4 \rightarrow s_1 \rightarrow s_2 \rightarrow s_{17}\) after using the maximum coverage rule. Tarantula only needs to examine four statements after using the maximum coverage rule.

According to Eq. (2), both \(d(t_3, t_6)\) and \(d(t_3, t_5)\) are equal to 5. The average distance between \(t_5\) and \(t_6\) is equal to 0. The average distance between \(t_3\) and other failed test cases is equal to 5, while the average distance between \(t_5\) and other failed test cases is equal to 2.5. The maximum distance rule and the minimum distance rule select \(t_3\) and \(t_5\) or \(t_6\), respectively. For the example, the maximum distance rule is as effective as the minimum coverage rule because the two rules rank \(s_3\) as the second. Similarly, the minimum distance rule is as effective as the maximum coverage rule. The rules can consistently improve the accuracy of SBFL based on Tarantula in terms of the example shown in Table 1.

4.3 Application of the Rules

If the program under test contains a fault, one of the rules is used only one time to mitigate the cost of spectrum-based fault localization. The process of fault localization using the rules only one time mainly includes three steps: Firstly, spectrum-based fault localization is used to rank all statements according to their suspicious values; Then, one of the rules is applied to select a failed test case and the potentially faulty statements can be identified based on coverage information of the selected test case; Finally, developers begin to examine every statement belonging to the potentially faulty statements according to their suspicious values and fix the faulty statement until the test case with respect to the faulty statement can be passed.

In general, the number of faults in the program under test is unknown. In this case, one of the rules could be applied for multiple times to localize all faults. Firstly, spectrum-based fault localization is used to rank all statements in the origin program under debugging. One of the rules is used to identify the potentially faulty statement set denoted as PFSS. Developers start to examine every statement belonging to PFSS one by one according to their suspicious values and fix the faulty statements. The statements having been examined are saved to a set called BENF in preparation for the application of one of the rules next time before all faults are localized. Note that the selected test case could detect more than one fault. In other words, the application of one of the rules only one time could localize more than one fault. Then, all test cases are executed on the fixed program. If all test cases can be passed, the process of fault localization stops. Otherwise, the following three steps are executed: Spectrum-based fault localization is used to rank all statements on the fixed program and PFSS is set to be empty; One of the rules is applied to identify the potentially faulty statement set; The statements belonging to PFSS and not belonging to BENF are examined one by one and the statements having been examined are further added
5. Experiment

We conducted the empirical studies on the Siemens suite to evaluate the impacts of five rules on 17 suspiciousness formulas. The experiment is performed on the same platform, i.e., Linux with GCC 4.8.5 and Windows 10 systems running Inter(R) Core i5 CPU 2.7GHz with 16GB physical memory. All localization techniques are developed based on the Java language and run on Windows system. The profile information is collected by a instrumentation tool gcov on Linux system.

5.1 Subject Programs

We conducted the empirical studies on the Siemens suite [30], which is still very popular to validate the effectiveness of different fault localization techniques [10]. Table 3 presents the subject programs including the name, the lines of code, a brief description of the functionality, the number of test cases, and the number of faulty versions. Some program versions are eliminated because the faults in those versions can be detected by none of test cases, e.g., version v32 of replace and version v9 of schedule2. Likewise, we also eliminated the versions that cause the exception of testing, e.g., version v27 of replace and version v1 of schedule. All in all, we selected 114 faulty versions of the Siemens suite. Note that each faulty version only contains a bug and the bugs are mutation-based artificially injected bugs.

5.2 Evaluation Metric

The Expense score [17] is widely used to evaluate the performance of a software fault localization technique in the SBFL community. The Expense score is the percentage of statements in the program under debugging that has to be examined when the first faulty statement is detected. The Expense score is defined as follows:

\[
\text{Expense score} = \frac{N_{\text{exam}}}{N_{\text{total}}} \times 100\%,
\]

where \(N_{\text{exam}}\) and \(N_{\text{total}}\) represent the number of statements examined and total number of statements in the program, respectively. A lower Expense score indicates a better performance.

5.3 Experimental Result and Analysis

The impact of a rule on SBFL is evaluated by the differences of the Expense scores before and after using the rule on SBFL. A value of 0 represents that the rule has no impact on SBFL. A positive value shows that the rule can effectively improve the accuracy of SBFL. On the contrary, the rule can impair the performance of SBFL. The larger the difference of the Expense score is, the better the improvement of fault localization accuracy is after using the rules.

Suppose \(X = \{x_i\}_{1 \leq i \leq n}\) is a set of the Expense scores before applying a rule in a given program, where \(i\) and \(n\) are version number and the number of faulty versions, respectively. And \(Y = \{y_j\}_{1 \leq j \leq n}\) is a set of the Expense scores after applying a rule in a given program, where \(j\) is version number. Since a subject program contains more than one faulty version, we calculated the difference of the Expense scores across all faulty versions of a given program after and before using a rule. The difference of the Expense scores after and before using a rule is defined as follows:

\[
\text{Diff} = \frac{1}{n} \sum_{k=1}^{n} (y_k - x_k).
\]

Table 4, Table 5, Table 6, Table 7 and Table 8 show the differences in Expense scores after and before using random, the maximum coverage and the minimum coverage, the maximum distance and the minimum distance rules, respectively.

Having seen Table 4, Table 5, Table 6, Table 7, and Table 8, we found that the differences in Expense scores after and before using the rules on 17 SBFL techniques are always greater or equal to 0. This indicates that the rules aid to improve the performance of SBFL. Compared with the localization techniques using Naish, Ochiai, and Tarantula before using the rules, the average costs of fault localization have decreased by more than 10% after using the rules. Particularly, the average costs of fault localization of Scott, M1, Wong2, and Fleiss across the subject programs have decreased by more than 42% after using one of the rules.

To examine the validity of the proposed rules, we conducted the statistical hypothesis testing. Since the number of faulty versions a part of subject programs contained is

| Name     | Lines of code | Test suite size | Number of faulty version | Brief description                  |
|----------|---------------|-----------------|--------------------------|-----------------------------------|
| tcas     | 173           | 1608            | 41                       | Altitude separation               |
| totinfo  | 406           | 1052            | 23                       | Information measure               |
| replace  | 563           | 5542            | 32                       | Pattern recognition               |
| schedule | 412           | 2650            | 9                        | Priority scheduler                |
| schedule2| 307           | 2710            | 10                       | Priority scheduler                |
| print_tokens | 565    | 4130            | 7                        | Lexical analyzer                  |
| print_tokens2 | 510        | 4115            | 10                       | Lexical analyzer                  |
| SBFL       | Subject Program | tact | totinfo | replace | schedule | schedule2 | print_tokens | print_tokens2 |
|------------|-----------------|------|--------|---------|----------|-----------|--------------|---------------|
| AMPLE      |                 | 23.7 | 6.2    | 4.3     | 14.0     | 27.5      | 16.8         | 1.2           |
| AMPLE2     |                 | 20.9 | 4.2    | 3.9     | 13.6     | 24.7      | 15.2         | 0.6           |
| Arithmetic Mean |           | 21.0 | 4.7    | 3.8     | 13.6     | 24.5      | 16.6         | 0.3           |
| Cohen      |                 | 21.0 | 5.0    | 3.7     | 13.9     | 24.4      | 17.0         | 1.2           |
| Fleiss     |                 | 53.2 | 64.2   | 42.7    | 57.4     | 50.2      | 14.9         | 24.6          |
| Jaccard    |                 | 21.0 | 4.7    | 3.1     | 6.9      | 19.6      | 19.2         | 1.2           |
| Kulczynski1|                 | 21.0 | 4.3    | 2.0     | 7.0      | 20.0      | 18.5         | 0.2           |
| Kulczynski2|                 | 22.6 | 3.6    | 1.9     | 7.3      | 20.2      | 16.3         | 0.3           |
| M1         |                 | 55.2 | 63.5   | 59.5    | 57.4     | 51.3      | 48.6         | 43.5          |
| M2         |                 | 20.8 | 3.8    | 1.7     | 6.9      | 18.3      | 17.2         | 1.2           |
| Naish      |                 | 20.7 | 3.5    | 7.7     | 14.6     | 19.2      | 16.0         | 2.5           |
| Ochiai     |                 | 22.6 | 3.7    | 2.0     | 7.1      | 19.8      | 17.3         | 0.3           |
| Scott      |                 | 53.2 | 63.5   | 40.5    | 57.2     | 49.5      | 15.0         | 23.3          |
| Tarantula  |                 | 22.8 | 5.0    | 2.8     | 7.2      | 20.8      | 16.5         | 0.3           |
| Wong1      |                 | 20.7 | 3.5    | 1.7     | 7.0      | 19.3      | 17.8         | 0.2           |
| Wong2      |                 | 55.0 | 64.3   | 59.4    | 57.4     | 49.6      | 48.6         | 44.3          |
| Wong3      |                 | 24.3 | 7.3    | 7.9     | 7.0      | 18.7      | 17.3         | 0.3           |

| SBFL       | Subject Program | tact | totinfo | replace | schedule | schedule2 | print_tokens | print_tokens2 |
|------------|-----------------|------|--------|---------|----------|-----------|--------------|---------------|
| AMPLE      |                 | 23.6 | 5.4    | 4.0     | 13.8     | 25.4      | 15.9         | 0.4           |
| AMPLE2     |                 | 21.0 | 3.8    | 3.3     | 13.5     | 23.1      | 14.9         | 0.2           |
| Arithmetic Mean |           | 21.0 | 3.8    | 3.3     | 13.5     | 23.1      | 15.0         | 0.2           |
| Cohen      |                 | 21.0 | 4.1    | 3.4     | 13.5     | 23.1      | 15.0         | 0.2           |
| Fleiss     |                 | 53.2 | 62.3   | 42.5    | 57.0     | 48.8      | 14.6         | 21.3          |
| Jaccard    |                 | 21.0 | 3.9    | 1.7     | 6.8      | 17.7      | 15.7         | 0.2           |
| Kulczynski1|                 | 21.0 | 3.9    | 1.7     | 6.8      | 17.7      | 15.7         | 0.2           |
| Kulczynski2|                 | 22.6 | 3.4    | 1.8     | 7.1      | 19.0      | 15.9         | 0.1           |
| M1         |                 | 54.9 | 62.3   | 58.3    | 57.0     | 48.8      | 48.3         | 42.2          |
| M2         |                 | 20.8 | 3.4    | 1.5     | 6.8      | 17.7      | 15.7         | 0.1           |
| Naish      |                 | 20.7 | 3.4    | 7.4     | 14.2     | 17.7      | 15.2         | 5.6           |
| Ochiai     |                 | 22.7 | 3.7    | 1.9     | 7.1      | 19.0      | 15.9         | 0.1           |
| Scott      |                 | 53.2 | 62.3   | 40.0    | 57.0     | 48.8      | 14.7         | 21.4          |
| Tarantula  |                 | 22.9 | 4.3    | 2.1     | 7.1      | 19.0      | 15.9         | 0.2           |
| Wong1      |                 | 20.7 | 3.4    | 1.4     | 6.8      | 17.7      | 15.7         | 0.1           |
| Wong2      |                 | 54.9 | 62.3   | 58.3    | 57.0     | 48.8      | 48.3         | 42.2          |
| Wong3      |                 | 24.3 | 7.1    | 7.5     | 6.8      | 17.4      | 14.9         | 0.1           |

| SBFL       | Subject Program | tact | totinfo | replace | schedule | schedule2 | print_tokens | print_tokens2 |
|------------|-----------------|------|--------|---------|----------|-----------|--------------|---------------|
| AMPLE      |                 | 23.9 | 9.2    | 6.5     | 14.3     | 32.0      | 25.1         | 5.8           |
| AMPLE2     |                 | 20.9 | 6.1    | 5.4     | 13.9     | 27.7      | 21.5         | 5.5           |
| Arithmetic Mean |           | 20.9 | 5.8    | 5.7     | 13.8     | 26.4      | 21.5         | 5.8           |
| Cohen      |                 | 20.9 | 7.0    | 6.2     | 14.2     | 27.7      | 22.9         | 6.2           |
| Fleiss     |                 | 53.2 | 69.2   | 46.5    | 58.6     | 53.7      | 16.7         | 27.7          |
| Jaccard    |                 | 21.0 | 6.8    | 4.6     | 7.3      | 23.1      | 25.8         | 6.1           |
| Kulczynski1|                 | 21.0 | 6.8    | 4.6     | 7.3      | 23.1      | 25.8         | 6.1           |
| Kulczynski2|                 | 22.6 | 4.3    | 4.0     | 7.8      | 21.6      | 25.1         | 2.6           |
| M1         |                 | 54.9 | 69.0   | 62.4    | 58.6     | 53.6      | 50.6         | 49.3          |
| M2         |                 | 20.8 | 4.7    | 3.8     | 7.0      | 21.7      | 24.9         | 3.2           |
| Naish      |                 | 20.7 | 4.0    | 8.7     | 15.0     | 19.9      | 16.2         | 5.9           |
| Ochiai     |                 | 22.6 | 5.9    | 4.4     | 7.4      | 23.1      | 25.1         | 4.5           |
| Scott      |                 | 53.6 | 69.2   | 44.1    | 58.6     | 53.7      | 16.9         | 28.2          |
| Tarantula  |                 | 22.8 | 7.6    | 5.1     | 7.8      | 24.4      | 26.8         | 6.1           |
| Wong1      |                 | 20.7 | 4.2    | 3.6     | 7.5      | 20.3      | 24.9         | 2.1           |
| Wong2      |                 | 55.3 | 69.0   | 62.4    | 58.6     | 53.6      | 50.6         | 49.3          |
| Wong3      |                 | 24.3 | 8.0    | 9.7     | 7.5      | 20.0      | 24.1         | 2.1           |
too small, no suitable hypothesis testing method is available. For example, program schedule only contain 6 faulty versions. Program print_tokens and print_tokens2 also contain a small number of faulty versions. In order to conduct the statistical hypothesis testing, we repeated every rule 50 times for every faulty version to generate sufficient samples. Similarly, every naive method was also repeated 50 times for every faulty version. We used the nonparametric Mann-Whitney U-test [31] to analyze the statistical differences between the rules and the naive methods. A null hypothesis is formulated to state that there is no statistical difference between the rule and the naive method at a significance level \( \alpha = 0.05 \).

For every faulty version, the fault type and the number of test cases detecting the fault is different. Therefore, for every program, we calculated the number of faulty versions where the rule is significantly superior to the naive method. Finally, we reported the sum of the number of faulty versions having the statistical differences. The result is shown as Table 9, where Rd, MaxC, MinC, MaxD, and MinD represent random rule, the maximum coverage rule, the minimum coverage rule, the maximum distance rule, and the minimum distance rule, respectively.

The improvement of the performance varies with the rules. The minimum coverage rule is consistently superior to or equal to the maximum coverage in terms of fault localization cost. The result in Table 9 also shows that the conclusion is hold. Similarly, the minimum distance rule is always better than or comparable with the maximum distance rule. The minimum coverage rule and the minimum distance rule have better or equal performances than other rules in terms of fault localization cost. The two rules can reduce the number of the statements examined by an average of more 20% across the subject programs. In terms of the number of faulty versions having statistical differences versus the naive methods, the two rules are superior to or equal to other rules. The main reason is that the number of the potentially faulty statements identified by the two rules are less

| SBFL       | Subject Program | tcas | totnfo | replace | schedule | schedule2 | print_tokens | print_tokens2 |
|------------|-----------------|------|--------|---------|----------|-----------|--------------|--------------|
| AMPLE      | 23.6%           | 8.4% | 4.5%   | 14.1%   | 32.5%    | 15.9%     | 0.4%         |
| AMPLE2     | 20.9%           | 5.3% | 3.2%   | 13.8%   | 27.6%    | 14.9%     | 0.0%         |
| Arithmetic Mean | 20.9%    | 5.0% | 3.4%   | 13.8%   | 26.3%    | 15.0%     | 0.0%         |
| Cohen      | 20.9%           | 6.0% | 3.6%   | 13.8%   | 27.6%    | 15.0%     | 0.0%         |
| Fleiss     | 53.3%           | 68.1%| 43.6%  | 57.9%   | 53.7%    | 14.6%     | 21.3%        |
| Jaccard    | 20.9%           | 5.8% | 2.0%   | 7.1%    | 23.0%    | 15.7%     | 0.0%         |
| Kulczynski1| 20.9%           | 5.8% | 2.0%   | 7.1%    | 23.0%    | 15.7%     | 0.0%         |
| Kulczynski2| 22.6%           | 4.2% | 1.9%   | 7.8%    | 21.5%    | 15.9%     | 0.1%         |
| M1         | 55.0%           | 67.9%| 59.6%  | 57.9%   | 53.6%    | 48.3%     | 42.2%        |
| M2         | 20.7%           | 4.4% | 1.6%   | 7.0%    | 21.6%    | 15.7%     | 0.0%         |
| Naish      | 20.7%           | 3.9% | 8.1%   | 15.0%   | 19.8%    | 15.2%     | 5.6%         |
| Ochiai     | 22.6%           | 5.2% | 2.1%   | 7.3%    | 22.9%    | 15.9%     | 0.0%         |
| Scott      | 53.3%           | 68.1%| 41.1%  | 57.9%   | 53.7%    | 14.7%     | 21.3%        |
| Tarantula  | 22.8%           | 6.4% | 2.5%   | 7.4%    | 24.3%    | 15.9%     | 0.0%         |
| Wong1      | 20.7%           | 4.2% | 1.6%   | 7.5%    | 20.2%    | 15.7%     | 0.1%         |
| Wong2      | 55.0%           | 67.9%| 59.6%  | 57.9%   | 53.6%    | 48.3%     | 42.2%        |
| Wong3      | 24.3%           | 7.9% | 7.6%   | 7.5%    | 19.9%    | 14.9%     | 0.1%         |

Table 7  The differences in the expense scores after and before using the maximum distance rule on 17 SFBL techniques

| SBFL       | Subject Program | tcas | totnfo | replace | schedule | schedule2 | print_tokens | print_tokens2 |
|------------|-----------------|------|--------|---------|----------|-----------|--------------|--------------|
| AMPLE      | 24.0%           | 7.2% | 5.7%   | 14.4%   | 26.5%    | 23.1%     | 5.7%         |
| AMPLE2     | 20.9%           | 5.7% | 4.8%   | 13.8%   | 23.8%    | 20.1%     | 5.6%         |
| Arithmetic Mean | 21.0%    | 5.8% | 5.1%   | 13.8%   | 23.7%    | 20.2%     | 5.8%         |
| Cohen      | 21.0%           | 6.7% | 5.7%   | 13.9%   | 23.8%    | 21.5%     | 6.3%         |
| Fleiss     | 53.6%           | 65.2%| 46.3%  | 58.0%   | 49.5%    | 16.9%     | 27.9%        |
| Jaccard    | 21.0%           | 6.0% | 3.9%   | 7.1%    | 18.6%    | 23.8%     | 6.2%         |
| Kulczynski1| 21.0%           | 6.0% | 3.9%   | 7.1%    | 18.6%    | 23.8%     | 6.2%         |
| Kulczynski2| 22.6%           | 3.7% | 3.2%   | 7.2%    | 19.6%    | 23.1%     | 2.5%         |
| M1         | 55.2%           | 64.9%| 62.3%  | 58.0%   | 49.3%    | 50.7%     | 49.3%        |
| M2         | 20.8%           | 4.0% | 2.9%   | 6.9%    | 18.3%    | 22.9%     | 3.1%         |
| Naish      | 20.7%           | 3.6% | 8.3%   | 14.5%   | 18.0%    | 15.5%     | 5.9%         |
| Ochiai     | 22.7%           | 5.1% | 3.5%   | 7.3%    | 19.7%    | 23.1%     | 4.6%         |
| Scott      | 53.6%           | 65.2%| 43.9%  | 58.0%   | 49.5%    | 17.0%     | 28.3%        |
| Tarantula  | 22.8%           | 6.8% | 4.4%   | 7.4%    | 19.9%    | 24.8%     | 6.3%         |
| Wong1      | 20.7%           | 3.6% | 2.5%   | 6.9%    | 18.3%    | 22.9%     | 2.1%         |
| Wong2      | 55.2%           | 64.9%| 62.3%  | 58.0%   | 49.3%    | 50.7%     | 49.30%       |
| Wong3      | 24.3%           | 7.4% | 8.7%   | 6.9%    | 18.0%    | 22.1%     | 2.1%         |

Table 8  The differences in the expense scores after and before using the minimum distance rule on 17 SFBL techniques
than or equal to those of other rules. The less the number of the potentially faulty statements is, the more the statements in the ranking list generated by SBFL are eliminated. In an extreme case where only a statement is executed by the failed test case identified by the two rules, after the application of the two rules, the statement can be always ranked first in the final ranking list, no matter what the ranking of the statement in the ranking list generated by SBFL is. The minimum coverage rule is slightly superior to the minimum distance rule in terms of statistically significant differences versus the naive methods because the number of faulty versions having statistical difference of the former is a bit more than that of the latter.

The minimum coverage rule and the maximum coverage rule determine the lower and upper bounds of the number of the statements executed by all failed test cases, respectively. The number of the statements executed by the test case determined by random rule falls out of the scope of the lower and upper bounds. Therefore, the improvement of random rule also lies between the improvements of the two rules. Comparing Table 4 with Table 5 and Table 6, we can draw the same conclusion. Similarly, in terms of the number of faulty versions having statistical differences versus the naive methods, random rule is slightly superior to the maximum coverage rule and worse than the minimum coverage rule.

5.4 Discussion

The proposed rules provide various ways to select a failed test case. Moreover, the rules can effectively improve the accuracy of spectrum-based fault localization. In particular, for the faults difficult to locate, i.e., SBFL needs to examine a large number of statements to detect the faults, the number of examined statements can significantly decrease after using the rules. For example, the best SBFL techniques need to examine 73 statements to detect the fault in version v41 of tcas. For the same fault, the numbers of examined statements after using the rules in the worse case and in the best case are 31 and 1, respectively. However, for the faults easy to locate, i.e., the statements containing the faults can be ranked at the top by SBFL without using the rules, the rules cannot effectively improve the rankings of the faulty statements. For instance, SBFL techniques with exception of SBFL using Wong1 consistently rank the faulty statement in version v2 of print_tokens2 second. After using the rules, the faulty statement is also ranked second. Although the rules cannot significantly improve the performance of SBFL in some cases, particularly for the faults easy to locate, the implementation of the rules on SBFL is well worthwhile because the rules can be applied with little cost.

Compared with the minimum coverage rule, the minimum distance rule needs to calculate the distance between pair-wise test cases. Although the calculation cost of the distance between pair-wise test cases is small, this slightly adds the additional cost to SBFL. Therefore, the minimum coverage rule is more effective for improving the accuracy of SBFL techniques using suspiciousness formulas in terms of the cost of fault localization. This indicates that the application of the test case having the minimum coverage for fault localization is more effective.

6. Threats to Validity

Although the experiment was carefully designed and operated, the study is not free from threats to its validity. These potential threats to validity can be summarized as follows.

Construct validity: A major threat to the construct validity is from the measure evaluating the performance of the rules. The application of other measures may generate different results. As previous studies, the Expense score is used to evaluate the effectiveness of SBFL using the rules. The impacts of the rules on SBFL are reflected by the difference of the Expense scores between after and before using the rules on SBFL. The metric could not adequately reflect the impacts. Additionally, as previous studies, our study was constructed based on two assumptions: One is that for any test case, the testing result is either fail or successful without other exceptions; The other is that the test suite contains a failed test case at least.

External validity: Our conclusion only depends on the Siemens suite written in C language because the suite is widely used in previous studies related to spectrum-based fault localization. Although the programs in the Siemens suite are from different application domains with different features, e.g., different number of faults, different lines of code, and different sizes of test suites, they could not be representative of other programs. The threat can be addressed by a further study on much more representative open-source and close-source softwares with the real faults, rather than synthetic faults.

Internal validity: SBFL utilize coverage information of program, as well as testing execution results. Coverage information is acquired by a instrumentation tool gcov. If the program includes unknown faults with the exception of the
seeded faults, the validity of coverage information could be threatened by unknown faults. The threat can slightly impair our conclusions. Additionally, the coincidental correctness test cases can also impair our conclusions.

7. Conclusion and Future Work

SBFL as a promising localization technique aims to reduce the cost of program debugging. The technique utilizes program spectrum information and testing running results to calculate the suspiciousness value of each program entity. The accuracy of SBFL is not always as satisfactory as expected because it neglects the contextual information of statements execution. Therefore, we proposed five rules to improve the accuracy of SBFL for further. One of failed test cases can be identified according to one of the rules. Only the suspiciousness values of the statements belonging to the program spectrum of the selected test case, rather than all statements, are calculated by a certain suspiciousness formula. The program spectrum of the selected test case can effectively preserve the contextual information of the statements execution. We designed and conducted the experiment to validate the impacts of the rules on 17 SBFL techniques. The results indicate that all five rules can significantly reduce the cost of SBFL. The minimum coverage rule is more effective for spectrum-based fault localization.

For future, our study will be extended to large scale industrial projects to validate the effectiveness of the rules. Moreover, we will also assign the weight to each statement covered by failed test cases according to their coverage capabilities. The weight information will be implemented on the existing suspiciousness formulas to further improve the effectiveness of SBFL. The relation between the coverage capabilities of failed test cases and the improving accuracy of SBFL will also be empirically investigated. Last but not least, we will consider context information of program to improve the accuracy of SBFL for further. The experiment results and source code are available at https://github.com/AlTestingKing/HRSBFL.

Acknowledgments

This work was supported by the Fundamental Research Funds for the Central Universities (Grant No. 2017XKQY081). The authors thank the anonymous peer reviewers for their comments and suggestions.

References

[1] J.S. Collofello and S.N. Woodfield, “Evaluating the effectiveness of reliability-assurance techniques,” Journal of Systems and Software, vol.9, no.3, pp.191–195, 1989.
[2] W.E. Wong, V. Debrov, and X. Du, “Towards better fault localization: a crosstab-based statistical approach,” IEEE Transaction on Systems Man & Cybernetics Part C, vol.42, no.3, pp.378–396, 2012.
[3] J.A. Jones, M.J. Harrold, and J. Stanko, “Visualization of test information to assist fault localization,” Proc. 24th International Conference on Software Engineering, Orlando, USA, pp.467–477, May 2002.
[4] R. Abreu, P. Zoeteweij, and A.J.C. Van Gemund, “An evaluation of similarity coefficients for software fault localization,” Proc. 12th Pacific Rim International Symposium on Dependable Computing, Riverside, USA, pp.39–46, Dec. 2006.
[5] W.E. Wong, V. Debrov, and B. Choi, “A family of code coverage-based heuristics for effective fault localization,” Journal of System and Software, vol.82, no.2, pp.188–208, 2010.
[6] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, “Spectrum-based fault localization in software product lines,” Information and Software Technology, vol.100, pp.18–31, 2018.
[7] Y. Guo, N. Li, J. Olluff, and A. Motro, “Exoneration-based fault localization for SQL predicates,” Journal of Systems and Software, vol.147, pp.230–245, 2019.
[8] X. Xie, T.Y. Chen, F.-C. Kuo, and B. Xu, “A theoretical analysis of the risk evaluation formulas for Spectrum-based fault localization,” ACM Trans. Softw. Eng. Methodol., vol.22, no.4, pp.1–40, 2013.
[9] M.J. Harrold, G. Rothermel, R. Wu, and Y. Li, “An empirical investigation of program spectra,” Proc. 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Montreal, Canada, pp.83–90, 1998.
[10] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault analysis,” IEEE Trans. Softw. Eng., vol.42, no.8, pp.707–740, 2016.
[11] H.A. De Souza, D. Mutti, M.L. Chaim, and F. Kon, “Contextualizing spectrum-based fault localization,” Information and Software Technology, vol.94, pp.245–261, 2018.
[12] J.M. Voas, “PIE: a dynamic failure-based technique,” IEEE Trans. Softw. Eng., vol.18, no.8, pp.717–727, 1992.
[13] R. Abreu, P. Zoeteweij, and A.J.C. Van Gemund, “On the accuracy of spectrum-based fault localization,” Proc. Testing: Academic and Industrial Conference Practice and Research Techniques, pp.89–98, 2007.
[14] T. Reps, T. Ball, M. Das, and J. Larus, “The use of program profiling for software maintenance with applications to the year 2000 problem,” Proc. 6th European Software Engineering Conference held jointly with 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Zurich, Switzerland, vol.22, no.6, pp.432–449, Sept. 1997.
[15] A.-B. Taha, S.M. Thebaut, and S.-S. Liu, “An approach to software fault localization and revalidation based on incremental data flow analysis,” Proc. International Conference on Software and Application, pp.527–534, 1998.
[16] H. Agrawal, R.A. Demillo, and E.H. Spafford, “An execution backtracking approach to program debugging,” IEEE Softw., vol.8, no.5, pp.21–25, 1991.
[17] J.A. Jones and M.J. Harrold, “Empirical evaluation of the tarantula automatic fault-localization technique,” Proc. 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, USA, pp.273–282, Nov. 2005.
[18] M. Reniers and S.P. Reiss, “Fault localization with nearest neighbor queries,” Proc. 18th International Conference on Automated Software Engineering, Montreal, Canada, pp.30–39, Oct. 2003.
[19] V. Dalimeier, C. Lindig, and A. Zeller, “Lightweight bug localization with AMPLE,” Proc. 6th International Symposium on Automated Analysis-Driven Debugging, Monterey, Canada, pp.99–104, Sept. 2005.
[20] W.E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, “Effective fault localization using code coverage,” Proc. 31st Annual International Conference on Computer Software and Application, Beijing, China, pp.449–456, July 2007.
[21] R. Abreu, P. Zoeteweij, R. Golstein, and A.J.C. Van Gemund, “A practical evaluation of spectrum-based fault localization,” Journal of System and Software, vol.82, no.11, pp.1780–1792, 2009.
[22] S. Yoo, X. Xie, F. Kuo, T.Y. Chen, and M. Harman, “No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist,” Technical Report, University College London, 2014.
[23] H.J. Lee, L. Naish, and K. Ramamohanarao, “Study of the relationship of bug consistency with respect to performance of spectra metrics,” Proc. 2nd IEEE International Conference on Computer Science and Information Technology, Beijing, China, pp.501–508, Aug. 2009.

[24] L. Naish, H.J. Lee, and K. Ramamohanarao, “A model for spectra-based software diagnosis,” ACM Trans. Softw. Eng. and Methodol., vol.22, no.3, pp.11:1–11:32, 2011.

[25] T.Y. Chen, F.-C Kuo, R. Merkel, and T.H. Tse, “Adaptive random testing: the art of test case diversity,” Journal of Systems and Software, vol.83, no.1, pp.60–66, 2010.

[26] H. Hemmati, A. Arcuri, and L. Briand, “Achieving scalable model-based testing through test case diversity,” ACM Trans. Softw. Eng. and Methodol., vol.22, no.1, pp.1–42, 2013.

[27] B. Jiang, Z. Zhang, W.K. Chan, and T.H. Tse, “Adaptive random test case prioritization,” Proc. 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, USA, pp.223–244, Nov. 2009.

[28] C. Fang, Z. Chen, K. Wu, and Z. Zhao, “Similarity-based test case prioritization using order sequence of program entities,” Softw. Qual. J., vol.22, no.2, pp.335–361, 2014.

[29] A. Podgurski, D. Leon, P. Francis, W. Marsri, M. Minch, J. Sun, and B. Wang, “Automated support for classifying software failure reports,” Proc. 25th International Conference on Software Engineering, Portland, USA, pp.465–475, May 2003.

[30] H. Do, S. Elbaum, and G. Rothermel, “Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact,” Empirical Software Engineering, vol.10, no.3, pp.403–435, 2005.

[31] F. Wilcoxon, “Individual comparisons by ranking methods,” Biometrics Bulletin, vol.1, no.6, pp.80–83, 1945.

Rongcun Wang  received the M.E. and Ph.D. degrees from Wuhan University of Technology and Huazhong University of Science and Technology in China in 2007 and 2015, respectively. He is currently an assistant professor at School of Computer Science and Technology, China University of Mining and Technology in China. His research interests include software testing, fault localization, and software maintenance.

Shujuan Jiang  received the B.E. degree from East China Normal University, Shanghai, China, in 1990, M.E. degree from China University of Mining and Technology, Xuzhou, China, in 2000, and Ph.D. degree from Southeast University, Nanjing, China, in 2006, respectively. She is currently a professor at School of Computer Science and Technology, China University of Mining and Technology. Her research interests include software testing, program analysis, fault localization, and test case generation.

Kun Zhang  He is currently a M.E. student at School of Computer Science and Technology, China University of Mining and Technology. He current research interests include software testing and fault localization.

Qiao Yu  received the Ph.D. degree from China University of Mining and Technology in 2017. She is currently a assistant professor at School of Computer Science and Technology, Jiangsu Normal University. Her current research interests include software analysis and testing, machine learning.