VFL: Variable-based fault localization

Jeongho Kim a,∗, Jindae Kim b, Eunseok Lee a,∗

a Sungkyunkwan University, Snoba-ro, Jangan-gu, Suwon, Republic of Korea
b HKUST, Clear Water Bay, Sai Kung, Hong Kong

ARTICLE INFO

KEYWORDS:
Software debugging
Software testing
Spectrum-based fault localization
Variable-based fault localization
Suspicious variable

ABSTRACT

Context: Fault localization is one of the most important debugging tasks. Hence, many automatic fault localization techniques have been proposed to reduce the burden on developers for such tasks. Among them, Spectrum-based Fault Localization (SFL) techniques leverage coverage information and localize faults based on the coverage difference between the failed and passed test cases.

Objective: However, such SFL techniques cannot localize faults effectively when coverage differences are not clear. To address this issue and improve the fault localization performance of the SFL techniques, we propose a Variable-based Fault Localization (VFL) technique.

Method: The VFL technique identifies suspicious variables and uses them to generate a ranked list of suspicious source code lines. Since it only requires additional information about variables that are also available in the SFL techniques, the proposed technique is lightweight and can be used to improve the performance of existing the SFL techniques.

Results: In an evaluation with 224 real Java faults and 120 C faults, the VFL technique outperforms the SFL techniques using the same similarity coefficient. The average Exam scores of the VFL techniques are reduced by more than 55% compared to the SFL techniques, and the VFL techniques localize faults at a lower rank than the SFL techniques for about 73% of the 344 faults.

Conclusion: We proposed a novel variable-based fault localization technique for more effective debugging. The VFL technique has better performance than the existing techniques and the results were more useful for actual fault localization tasks. In addition, this technique is very lightweight and scalable, so it is very easy to collaborate with other fault localization techniques.

1. Introduction

Software debugging is an expensive software development task, which accounts for 50%–80% of the total software development costs [19,42]. This task consists of fault localization and fixing, especially fault localization is a very difficult task that takes more time and cost. In addition, since faults cannot be fixed if the location of faults are not found, fault localization task is very important.

Therefore, many researchers have proposed various fault localization techniques such as spectrum-based [32], mutation-based [28], and slice-based [41] techniques. Among these techniques, the Spectrum-based Fault Localization (SFL) technique is the most popular which takes 35% of all fault localization-related studies [17]. The SFL techniques provide a ranked list of suspicious entities (such as statements, functions, basic blocks, and classes) to help developers localize a fault and fix it during debugging.

Many SFL techniques with various similarity coefficients such as Tarantula [12], Jaccard [1], Naish [23], and GP [45] have been proposed to improve localization performance. The results and formulas of these similarity coefficients are different, but they are all designed for the same purpose. The intuition behind the SFL techniques is that if an entity covered by more failed tests and fewer passed tests is more likely to be faulty. However, such SFL techniques cannot localize faults effectively if the coverage information is not sufficient to distinguish the execution of failed and passed test cases [2,10]. In other words, performance is not guaranteed if there is not enough reliable data because the SFL techniques are statistical approaches. For example, suppose a program has one if statement. It is possible that a failed test case executes the then-block of the if statement, although it should execute the else-block. The problem occurs when a given passed test case executes the then-block just like the failed test case. Since both failed and passed test cases execute the same block, they have the same coverage. With such coverage information, the SFL techniques can only indicate that a fault is localized somewhere in the then-block. If there are many source code lines in this block, this outcome is not satisfactory. Moreover, given that the failed test case took the wrong branch, it is more likely that a fault exists in the statements before the branch, which leads the execution to the wrong side of the branch. Various techniques [10,18,33,34,46-48]
have been proposed to overcome this issue, and we wish to contribute a technique that addresses this problem differently.

Our idea to overcome this limitation and improve the fault localization performance of the SFL techniques is to leverage additional information about program entities. More precisely, we focus on variables that appear at the source code lines and try to find suspicious variables. Then, we consider source code lines that are closely related to suspicious variables as possible fault locations.

The intuition behind this idea is that developers often monitor the values of suspicious variables while debugging. If a variable has an unexpected value at a certain location, it is likely that a fault resides at the statements containing this suspicious variable. Therefore, a developer checks the locations of that variable to localize the fault. In this debugging scenario, finding suspicious variables and localizing a fault among statements where such variables appeared can be an effective approach for fault localization to improve the existing SFL techniques.

Hence, we propose a Variable-based Fault Localization (VFL) technique to improve the fault localization performance of the SFL techniques. The proposed the VFL technique only requires information about variables appearing in the source code and coverage information. Since many tools have already been developed, such as Ctags, Cscope and JavaParser, variable information can be easily obtained without additional human efforts. Also, coverage information can be easily obtained from inputs originally used in existing SFL techniques. Therefore, the VFL technique is a lightweight technique that can be combined with existing SFL techniques by simply borrowing their similarity coefficients.

- The detailed process of the VFL technique is as follows.
  1. Extract variables and function invocations information and then calculate the suspicious ratio of the individual variables and function invocations.
  2. Assign the suspicious ratio of individual statements that contain variables and function invocations using the suspicious variable and function invocations ranked list. (If a statement contains several variables and function invocations, the maximum value is selected.)
  3. Perform debugging in the order of a ranked list of statements derived from the above steps.

To evaluate the proposed technique, we applied the technique to 224 real faults from four Java programs in the Defects4j dataset [14] and 120 faults from seven C programs in the Siemens suite [30].

- The contributions of this paper are listed below.
  1. A novel variable-based fault localization technique with improved performance that can benefit existing SFL techniques.
  2. A new approach to fault localization based on suspicious variables that can be identified by information originally available to the SFL techniques.

The rest of the paper is organized as follows. First, we introduce techniques that are the background of the proposed technique in Section 2. Then, we describe the details of the proposed approach with a motivating example in Section 3. Then, we present research questions and an experimental setup for evaluation of the proposed approach in Section 4. In Section 5, we provide evaluation results of the proposed technique. Then, we discuss the issues of this paper in Section 6. After that, we discuss threats to the validity (Section 7) and related work (Section 8). Finally, we conclude the study with suggestions for future work in Section 9.

2. Background

The SFL technique identifies suspicious statements using execution traces and test results. This technique determines the order of statements requiring fixes based on a statistically computed suspicious ratio. Failed test cases increase the suspicious ratio of statements, while passed test cases reduce the suspicious ratio of statements that are relatively unrelated to program failure. Therefore, the SFL technique cannot be applied in the absence of at least one failed test case. The definition of the SFL technique is as follows:

**Definition 1.** Spectrum-based fault localization technique

Given a program P with n statements S = {s1, s2, s3} and m test cases t = {t1, t2, t3, t4} vectors V = {Ntf, Ntp} can be calculated for each Sn using an SFL matrix (n × m) Similarity coefficients can be calculated using V.

The goal of the SFL technique is to provide a developer with faulty statements with a ranking of S for effective debugging tasks. The SFL technique utilizes four parameters based on statements, whether covered or not, and the test results (pass or fail). Ntf indicates the number of failed test cases covering the statement, Ntp indicates the number of passed test cases not covering the statement, Ntp indicates the number of passed test cases covering the statement, and Ntp indicates the number of passed test cases not covering the statement. It computes the Ntf, Ntp, Ntp, Ntp of each statement based on the program spectra information and then calculates the suspicious ratio using the similarity coefficients. There are various similarity coefficients [4], but we compare our technique to the nine most up-to-date and representative techniques described in Table 1.

### Table 1

<table>
<thead>
<tr>
<th>Similarity coefficient</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>[\frac{N_{tf}}{N_{tf} + N_{tp}}]</td>
</tr>
<tr>
<td>Jaccard</td>
<td>[\frac{N_{tf}}{N_{tf} + N_{tp} + N_{tp}}]</td>
</tr>
<tr>
<td>Naïve</td>
<td>[N_{tf} - \frac{N_{tp}}{N_{tp} + N_{tp}}]</td>
</tr>
<tr>
<td>GP08</td>
<td>[N_{tf}^2(2N_{tp} + 3N_{tp} + 3N_{tp})]</td>
</tr>
<tr>
<td>GP10</td>
<td>[\sqrt{N_{tf} + N_{tp}}]</td>
</tr>
<tr>
<td>GP11</td>
<td>[\sqrt{N_{tf} + N_{tp} + N_{tp}}]</td>
</tr>
<tr>
<td>GP13</td>
<td>[\frac{N_{tf}(2N_{tp} + 2N_{tp} + 3N_{tp})}{N_{tp} + N_{tp}}]</td>
</tr>
<tr>
<td>GP20</td>
<td>[N_{tf} + N_{tp} + N_{tp}]</td>
</tr>
<tr>
<td>GP26</td>
<td>[2N_{tf} + 2N_{tp} + N_{tp}]</td>
</tr>
</tbody>
</table>

3. Approach

In this section, we present a motivating example that explains why a variable-based approach can produce better results than previous line-based approaches and introduce our variable-based fault localization technique.

3.1. Motivating example

The SFL technique is a technique that statistically assigns suspicious ratios to entities based on the execution traces and test results of a test suite. Although there are differences in the similarity coefficients, the SFL technique basically assigns the entity a high suspicious ratio if the failed test case is highly covered and the passed test case is less covered [5,35]. Therefore, even if a suspicious ratio is given at the statement level, which is the finest-grained entity, all the entities included in the same branch are given the same suspicious ratio. Of course, this problem also occurs for other entities and all of them are given the same rank. This phenomenon is generally defined as a tied-rank problem.

Fig. 1 indicates the tied-rank problem of the traditional SFL technique using the binary search algorithm source code as an example. This algorithm determines the position of a particular value in ascending order in an ordered array. The algorithm selects the first intermediate value and compares the value with that to be searched. If the first selected
if (start < end) {
    mid = (start + end) / 2;
    if (key == arr[mid]) { return mid; }
    else if (key < arr[mid]) { return binarySearch(arr, start, mid - 1, key); }
    else if (key > arr[mid]) { return binarySearch(arr, mid + 1, end, key); }
}

return -1;

public static int binarySearch(int[] arr, int start, int end, int key) {
    if (start > end) { return -1; }
    int mid = (start + end) / 2;
    if (key == arr[mid]) { return mid; }
    else if (key < arr[mid]) { return binarySearch(arr, start, mid - 1, key); }
    else if (key > arr[mid]) { return binarySearch(arr, mid + 1, end, key); }
}

Fig. 1. A buggy program and its test suite.

3.2. Variable-based fault localisation

In this section, we introduce our variable-based fault localization technique in detail. Fig. 2 illustrates the entire process of the VFL technique. First, we construct a variable list by parsing the source code. Then, we execute the program with the test cases, and collect the variable spectra as the program is running. The variable spectra are calculated based on the execution traces and the test results of the program. Next, we calculate the suspicious ratio of the variables by substituting the variable spectra into the similarity coefficient. Finally, we obtain a suspicious variable ranked list that is sorted in descending order relative to the suspicious ratios of the variables.

The difference between the traditional SFL technique and the VFL technique is that they calculate suspicious ratios for different entities (i.e., at the statement level and variable level, respectively). That is, the SFL technique assigns suspicious ratios to individual statements based on the spectra covered by the test suite, while on the other hand, the VFL technique assigns suspicious ratios to individual variables, for which it then recalculates the suspicious ratio of each statement based on this ranked list of suspicious variables.

Algorithm 1 describes the entire algorithm of the VFL technique in detail. The inputs are a program, a set of failed test cases, a set of passed test cases, and a similarity coefficient, and the output is a ranked list of
suspicious statements. In the beginning, the set of variables declared in the program is initialized to V, and the source code line of program is initialized to L. In the loop statements 8 and 9, i, j repeat until V, L, respectively. The suspicious ratios are calculated to generate a ranked list of all variables in these loop statements.

First, we calculate the values of $N_{fp}$, $N_{tp}$, $N_{rp}$, and $N_{up}$ in statements 10–13. $N_{fp}$ is the number of test cases that covered variable V among the total failed test cases, $N_{tp}$ is the number of test cases that uncovered V among all failed test cases, $N_{rp}$ is the number of test cases that covered V among the total passed test cases, and $N_{up}$ assigns V as the number of uncovered test cases among all passed test cases. When the same variable is used several times in the source code, $N_{fp}$, $N_{tp}$, $N_{rp}$, and $N_{up}$ are assigned several times. In statement 14, the four parameters are grouped into a tuple $V_i(N_{tuple})$. Statement 15 uses a similarity coefficient to calculate the average suspicious ratio of the variables. Next, the order of the statements containing the same variable is rearranged in statement 16 because the same variables are assigned the same suspicious ratio. Specifically, this sorting satisfies that $N_{r}$ is large and $N_{i}$ is small. This is motivated by the fact that the suspicious ratios of the statements are reassigned using the suspicious ratios of the variables in statement 17.

Table 2 details how to calculate the suspicious ratio by applying the VFL technique to the buggy program in Fig. 1. First, we prepare a list of variables like second row of Table 2 by parsing the variables declared in the example source code. Then, both the suspicious ratio and the rank of these variables are obtained based on the execution traces and test results. The difference between the SFL and the VFL technique is that the SFL technique calculates the suspicious ratio of the statements and the VFL technique calculates the suspicious ratio of the variables. For example, the variable $arr$ is used in statements 5, 6, 7, and 8, and the suspicious ratios of these variables are 0.6, 0.6, 0.4, and 0.4, respectively, using the Jaccard correlation coefficient. We define the average suspicious ratio of these individual variables as the VFL Score. Therefore, the VFL Score of the $arr$ is 0.4 (($0.6 + 0.6 + 0.4 + 0.4)/4 = 0.5$). In this way, the VFL Scores of all declared variables are calculated and their rankings are also obtained. Finally, we calculate the Final VFL Rank based on these VFL Scores and the VFL Rank. The Final VFL Rank is used to select the variable with the highest VFL Rank and then order the suspicious ratios of the statements containing the variable in order. For example, the variable with the highest VFL Score in the example source code is $start$. The statements containing this start variable are 3, 4, and 6. Therefore, the Final VFL Rank 1 is assigned to statement 6, which has the highest suspicious ratio in the start variable. Then, we find the statement with the next highest suspicious ratio in the same variable. Since the ranks of statements 3 and 4 are the same, rank 2 is assigned to all of them in the Final VFL Rank. Next, we find end, which is the second highest VFL Score, and repeat the above operation. Statements with multiple variables can be given a repetitive Final VFL Rank. In such a case, the Final VFL Rank is determined by which variable is ranked first. That is, when various ranks are given, the highest rank is selected, and the rest are skipped.

Because the SFL technique simply uses execution traces and test results information, it results in the same suspicious ratio and rank for all statements with the same coverage. Therefore, a large amount of effort is needed to break the tied ranks, which is accomplished by extending the information or improving the SFL technique itself. Many studies have been conducted, as a solution to this problem can reduce the time and cost for developers to localize faults [48–50]. Therefore, the ultimate goal of this paper is to propose a VFL technique that can solve the tied-rank problem of the SFL technique.

The motivating example in Fig. 1 illustrates how to apply the SFL technique to calculate the SFL Rank. When debugging with the SFL technique, the ranks of statements 8, 9, 18, and 19, which comprise the

---

**Table 2**

Rank computation of suspicious variables.

<table>
<thead>
<tr>
<th>Line</th>
<th>Suspicious variable ranked list</th>
<th>Final VFL Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>arr start end key mid</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.57 2 0.57 1 0.57 3 2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.6 1 0.6 1 0.6 1 0.5 3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.6 1 0.6 1 0.4 3 0.4 4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.4 3 0.4 3 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>0.4 3 0.4 4 0.4 4 0.4 4</td>
<td></td>
</tr>
<tr>
<td>VFL Score</td>
<td>0.5</td>
<td>0.58</td>
</tr>
<tr>
<td>VFL Rank</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 2. An overview of the variable-based fault localization technique.
group that contains actual faults, are ranked first. However, because there are four statements with the same rank, the developer must check at least 1 to a maximum of 4 statements to localize the actual faults in the source code. On the other hand, when debugging with the VFL technique, the ranking of statement 8, which includes an actual fault, is the unique first rank in Table 2. Therefore, the VFL technique can break the tied ranks occurred in the SFL technique, thus improving the performance.

Because the SFL technique is highly dependent on the quality of the test cases, a small diversity of test cases cannot guarantee high accuracy. However, the above example is a toy program with enough test cases. Therefore, in this case, the accuracy of the SFL Rank is relatively high and the probability of localizing the fault is high. That said, real-world projects often fail to meet all of these conditions. Therefore, the performance difference between the SFL and the VFL techniques is more prominent when applied to real-world projects.

3.2.1. Handling statements without variables

Unlike the existing SFL techniques, the VFL technique calculates a suspicious ratio based on a variable, not a statement. This technique is designed with the assumption that most executable statements contain variables and that the probability that a statement without a variable is defective is very low, for which we have verified these assumptions through experiments. However, all faults do not necessarily occur in statements including variables. Therefore, in this chapter, we explain how the VFL technique can handle statements that do not contain variables.

To explain the VFL technique step by step, we classify the proposed technique into the VFL (V: Variable) and the VFL (VF: Variable and Function invocation) versions. The VFL (VF) technique is designed to enable the VFL (V) technique to handle additional statements that do not contain variables.

In other words, the VFL (V) technique assigns only suspicious ratios based on variables and VFL (VF) can handle not only statements including variables but also statements excluding variables. Therefore, the VFL (VF) technique is more accurate and scalable than the VFL (V) technique. Finally, the VFL technique proposed in this paper is the VFL (VF) technique and the VFL (V) is a concept to introduce the main idea of the VFL technique.

First, the type of statement is classified as assignment, condition, command, loop, and other statements. The assignment statement is used to assign values on the right-hand side to values of the variables on the left-hand side of equal signs. The condition statement is used to determine whether the dependent statements are executed by confirming that the condition is satisfied. This statement can be set to various conditions by adding else if and else statements. The command statement consists of an invocation and a return statement. The invocation statement is used to invoke an internal or external function with arguments. The return statement is used to return data such as a specific variable, array, or table from a function. The loop statement is used to iterate and execute dependent statements by determining the start and end points. Finally, the other statement consists of declaration and annotation statements. The declaration statement is used to define the type of variables and initialize values, and the annotation statement is an unexecutable statement that gives additional explanations of the source code, including functions, variables and so on [3,7,36].

Among the types of statements described above, the only one that does not contain variables is the command statement. The function invocation statement is used to invoke internal and external functions with arguments. Therefore, the variables can be included as an argument, but it is not mandatory, so the function invocation statement may not necessarily contain a variable. In other words, in the case of a function that does not require an argument, this invocation statement does not contain a variable. Therefore, if there is a fault in the function invocation statement, the accuracy of the VFL (V) technique can be lowered in specific cases. For example, suppose that there is a fault in the function Calendar.getInstance(getCurrentTimeZone()). Because this is a simple function invocation statement that gets the current time in the system, no arguments are needed. Therefore, it is difficult to localize this kind of fault with the VFL (V) technique.

Except for the function statement, the remaining statements such as assignment, condition, loop, and declaration statements generally use variables, as listed in Table 3.

However, some of these variables can be replaced with function invocation as in Table 4. That is, the value returned by the function invocation can be used instead of the value of the variable. Therefore, the example source code of Table 3 can be transformed like Table 4.

Because the VFL (V) technique assigns a suspicious ratio to each variable, it is very difficult to localize faults in function invocation statements that do not contain variables. Furthermore, the performance of the VFL (V) technique cannot be guaranteed if there are faults in other types of statements, including function invocation as described in Table 4. Therefore, it is necessary to additionally handle function invocation statements to improve the performance of the VFL (V) technique. In other words, we extend the VFL (V) technique to further consider the function invocation statements.

The difference between the VFL (V) and the VFL (VF) techniques is that the VFL (V) technique assigns suspicious ratios based on variables, while the VFL (VF) technique assigns suspicious ratios based on both variables and function invocation statements. That is, the VFL (V) technique calculates the suspicious ratio for each variable and then assigns suspicious ratios to the statements containing these variables, while the VFL (VF) technique calculates the suspicious ratio for each variable and function invocation statement and then assigns suspicious ratios to the statements containing these variables and function invocations. For example, suppose that the Main() function contains a statement that invokes the getCurrentMonth() function. The VFL (V) technique only assigns suspicious ratios to individual variables. Therefore, a suspicious ratio is not assigned to this function invocation statement because the statement that invokes getCurrentMonth() does not contain any variables. On the other hand, the VFL (VF) technique assigns suspicious ratios both to variables and function invocation statements. Thus, the VFL (VF) technique calculates suspicious ratios based on the execution traces and the test results of the statement that invokes the getCurrentMonth() function. When the same variable is repeatedly used in several statements, the VFL (V) technique collects all the information regarding the statements that include this variable, and then uses this as information to calculate the suspicious ratio of the variable. Similarly, if getCurrentMonth() is invoked multiple times, the VFL (VF) technique collects all the information regarding the invoked statements and immediately computes the suspicious ratio of getCurrentMonth().
Finally, the return statement, which is the remainder of the command statement, is used to return data of the function. If a fault occurs in this return statement, the suspicious ratio of the function also increases. Therefore, the VFL (VF) technique assigns a high suspicious ratio to the function including the return statement. In other words, performance improvement can also be expected by applying the VFL (VF) technique. In addition, since the annotation statement is not executed directly in the program, the description of this statement is skipped.

4. Evaluation

In this section, we present our research questions and experimental setup.

4.1. Research questions

4.1.1. RQ1: How well does the VFL (V) technique localize faults?

The main purpose of using an automatic fault localization technique is to help the developers during debugging by reducing the search area for finding a fault. Hence, we evaluated the VFL (V) technique by assessing the amount of source code to be examined until a real faulty statement is found.

4.1.2. RQ2: How well does the VFL (V) technique place faulty statements at low ranks?

Although answering RQ1 can provide information about the overall performance of the VFL (V) technique, we want to evaluate the VFL (V) technique more thoroughly. From RQ1, we obtain an answer such as ‘only 3% of statements are examined when a fault is found’. However, this is not helpful for developers if the examined program has over one million lines of code. Therefore, we apply more strict conditions (e.g., we examine only the top N statements of a ranked list) and determine whether the VFL (V) technique can localize a fault.

4.1.3. RQ3: Is the VFL (V) technique effective for every fault?

It is known that no techniques show the best performance for all bugs [11,15,42]. One fault localization technique can show impressive performance for some bugs, but perform poorly for other bugs. This same argument can be applied to the VFL and the SFL techniques. Hence, it is important to verify that the VFL (V) technique performs more effectively than the SFL technique for the majority of faults.

4.1.4. RQ4: Can we increase the VFL (V) technique performance and applicability by using additional techniques?

We introduce a technique to list suspicious variables to localize fault statements. However, a faulty statement might not contain any variables, and the VFL (V) technique cannot be applied in this case. Therefore, we extend the VFL (V) technique to enable handling of statements that do not contain variables. We name this technique as VFL (VF) in this paper and evaluate this performance through experiments.

4.2. Experimental setup

In this section, we present the experimental setup for evaluation of the proposed technique.

4.2.1. Benchmarks and Tools

To evaluate the proposed VFL technique, we employ the Defects4j dataset [14] for Java programs and the Siemens suite [30] for C programs as our sets of faults. The Defects4j dataset provides faults along with human-written patches, test cases, and other information useful for fault localization studies [9,14,22,43]. The Siemens suite is a popular set of faults that has been used for most fault localization studies [8,27,29,39,40]. Table 5 shows details related to the two datasets used in this study. The Defects4j dataset originally consisted of five different Java projects, but we excluded Closure since its test cases do not follow the standard JUnit structure, resulting in some difficulties obtaining coverage information. However, even for a partial dataset, we secured 224 real bugs from large software projects with at least 22,000 lines of code, and these were sufficient to conduct our study. The Siemens suite consists of 132 versions of seven C programs. However, only 120 identified faulty versions were used in this experiment. We excluded the other 12 versions since they have no fault or their faulty lines reside on header files that cannot be localized by standard SFL technique.

The proposed techniques require two types of data. Therefore, we use open source tools to collect this data automatically. The first type of data is the execution traces, which is necessary to calculate suspicious ratio by applying the existing SFL technique. This is a collection of statements covered by individual test cases. In this experiment, the Java-based Defects4j dataset uses Gzoltar and the C language-based Siemens suite uses Clover to collect this data. Second, the additional data needed to apply proposed VFL techniques are variable and function invocation information. Variable information refers to the names of variables declared in the program along with their location information. Function information refers to the names of functions coded in the program. In this experiment, the Java-based Defects4j dataset uses ASTParser and the C language-based Siemens suite uses Ctags to collect this data. We can collect all the data we need using a variety of open source tools. Therefore, these tasks are very simple and easy, not requiring any additional human effort. Further, variable names and function invocation information can be gathered from source code statically, as opposed to dynamically. Therefore, the time and cost of collecting variable names in order to apply the VFL (V) technique is negligible. In addition, to enable the VFL (V) technique to handle statements that do not contain variables, not only variable names but also function invocation information are needed. However, the time and cost of collecting this information is also negligible. Most open source projects do not require additional effort because these tools can automatically collect information as long as we specify the type of information we want.

4.2.2. Evaluation metric

One popular metric of evaluating the performance of fault localization techniques is an Exam score [13,21,24,26,31]. The Exam score indicates the amount of source code that needs to be examined until actual fault is localized. Since fault localization techniques often provide a ranked list of suspicious lines, we can assume that the developer examines each line in the given ranked list from the top. Once a developer finds a fault on an examined line, the rank of the line indicates the number of source code lines the developer already examined. Hence, we can compute the proportion of such examined lines to all source code lines to assess how well a fault localization technique localizes an actual fault location. The detailed formula to compute the Exam score is as follows.

\[ \text{Exam Score} = \frac{\text{Rank of an actual fault location}}{\text{Total number of code lines}} \times 100 \]

We consider two types of Exam scores: Best (B) and Worst (W). It is possible that multiple lines can have the same rank in a ranked list, and we need to decide how many lines of the same rank to examine. For example, suppose there are five lines with rank 7, and one of these lines is a faulty line. Since all lines above rank 6 have already been examined, we only need to confirm how many of the five lines at rank 7 are examined until the faulty line is found. If the faulty line is examined first among the rank 7 lines, this is the best case (e.g., Exam score (B)). On the contrary, if we assume that the faulty line is examined after all other rank 7 lines are examined, this is the worst case (e.g., Exam score (W)).

The Exam score is highly dependent on the sensitivity of the correlation coefficient to calculate the suspicious ratio. This sensitivity means the suspicious ratio increases when one failed test case is covered and decreases when one passed test case is covered. Of course, the sensitivities of the correlation coefficients are different because these are designed
for different purposes and environments. Therefore, a high-performance SFL technique is good for classifying the difference between failed test cases and passed test cases with suitable sensitivity. The higher the sensitivity of the correlation coefficient, the lower the Exam score (B) but the higher the Exam score (W). This means that although a fault is found in the upper ranking, there are many statements with the same ranking. For example, the correlation coefficient named Intersection [4] defines the suspicious ratio simply as the number of failed test cases covered.

Intersection = The number of covered failed test cases

Suppose a faulty program has one failed test case and a hundred passed test cases. If the Intersection correlation coefficient is applied to this program, the suspicious ratio of all statements covered by the failed test cases is 1, otherwise it is 0. Therefore, in this case, the Exam score (B) of this correlation coefficient is almost zero, but the Exam score (W) is almost 100. In addition, suppose that the Exam score (B) using the Tarantula correlation coefficient is 0.5 and the Exam score (W) is 5. The Exam score (B) of the Tarantula is higher than the Intersection, but not less accurate. The reason for this is that the Exam score (B) of Tarantula is worse than the Intersection, but the Exam score (W) is much better. Therefore, it is more accurate to compare the performance of the Exam score (W) than the Exam score (B). For this reason, we used a stricter metric, Exam score (W) in this study.

Exam score is most commonly used as a metric for comparing the performance of various types of fault localization techniques. However, this score may not help developers localize faults, practically speaking, because developers do not check so many statements to find faults. In other words, most of developers have complete trust in this kind of result from automated techniques [55]. Therefore, we also use Top N as an additional metric to answer RQ2. Top N is defined as localizing the actual faults in the top Nth ranked statement. We decided to examine the Top 1, 5, 10, 20, and 50 lines, since these numbers were used in a survey of developers conducted in a previous study. Developers prefer the Top 5 (73.58%) as the minimum criteria for debugging. The next rankings are Top 10 (15.09%), Top 1 (9.43%), and Top 20 (1.35%). There were very few developers who wanted to use the Top 50, but no single value was more than a minimum criterion for debugging [17]. Therefore, we evaluate the performance of the proposed technique based on Top 1, 5, 10, 20 and 50.

5. RESULTS

This section presents evaluation results and a corresponding analysis to answer the research questions introduced in Section 4.

5.1. RQ1: How much effort can be saved with the VFL (V) technique?

To answer RQ1, we computed the Exam Scores of the VFL (V) and the SFL techniques for all 344 faults from the Defects4j dataset and Siemens suite. Fig. 3 shows the Exam scores of the VFL (V) and the SFL techniques using nine different similarity coefficients. Each group represents the Exam scores of the SFL (left) and the VFL (V) (right) techniques computed with a similarity coefficient. The Exam scores shown here are the average scores of 344 programs in the Defects4j dataset and Siemens suite for each similarity coefficient. We also present the reduced effort when using the VFL (V) technique instead of the SFL technique as a line graph. For example, in Tarantula, the SFL Exam score is 7.51, the VFL (V) Exam score is 3.31, and the reduced effort is about 56%.

This result indicates that the SFL technique should examine 7.51% of the total source code lines to find the fault, while the VFL (V) technique can find faults by reviewing only 3.31% of the total lines. Therefore, our VFL (V) technique can reduce about 56% of the effort for examination compared to the SFL technique with Tarantula. Note that the reduced effort is dependent to the similarity coefficient used in the VFL and the SFL techniques. In addition, the maximum value is 56.41% in GP26, and the minimum value is 51.81% in GP11. The average scores of the SFL and the VFL (V) Exam scores are 6.87 and 3.11, respectively. The average reduced effort for all similarity coefficients is about 55%.

We further perform statistical significance tests to identify pre-post differences between accuracy of the SFL and the VFL (V) techniques. If the sample size is very small or if the sample data does not assume normality, this data should be tested with non-parametric techniques. Therefore, we use the Wilcoxon signed ranks test because our sample size is sufficient but the normality is not met. This is a statistical technique of comparing the mean of two variables ($\theta_1$, $\theta_2$). Therefore, it is mainly used to measure the effects of an experiment or treatment. We set the accuracy of the SFL technique to pre-data and the accuracy of the VFL (V) technique to post-data to prove that they are different. The null hypothesis and the alternative hypothesis are set as follows.

- $H_0$(null hypothesis) : $\theta_1 = \theta_2$ (There is no difference between the accuracy of the SFL technique and that of the VFL (V) technique.)
- $H_1$(alternative hypothesis) : $\theta_1 \neq \theta_2$ (There is a difference between the accuracy of the SFL technique and that of the VFL (V) technique.)

For this hypothesis test, we apply nine similarity coefficients to both the SFL and the VFL (V) techniques. The accuracy of the SFL technique used as the pre-data is defined as the ranking of the statements of the actual faults localized by applying the SFL technique. The accuracy of the VFL (V) technique used as the post-data is defined as the ranking of the statements of the actual faults localized by applying the VFL (V) technique. In addition, we use the two-tailed test rather than the one-tailed test for more rigorous hypothesis testing.

We apply the SFL technique to 224 Defect4j dataset programs and 120 Siemens suite programs to calculate the accuracy ($\theta_1$) for each program. We also calculate the accuracy ($\theta_2$) of the VFL (V) technique in the same way. The ultimate goal of this hypothesis test is to confirm by Wilcoxon signed ranks test that there is a difference between these two variables ($\theta_1$, $\theta_2$). In Table 6, since the asymptotic significance of all pairs is less than 0.01, the null hypothesis is rejected at a 99% significance level. This result implies that there is a clear difference between
the accuracy of the SFL technique and that of the VFL (V) technique using all similarity coefficients in the experiment.

We also calculated the r value of the Wilcoxon signed ranks test to obtain the effect size in Table 6. Generally, if the r value is greater than 0.1, 0.3, or 0.5, each effect is defined as small, medium, or large size. In the experimental results, although the effect is medium size in the case of Jaccard, GP13 and GP26, the effect is large size in the remaining cases. In addition, since the average of these is 0.511, the average effect is large size. In other words, the average effect size of the SFL technique and the VFL (V) technique is large. Therefore, we can conclude that the VFL (V) technique has improved much more than the SFL technique.

5.2. RQ2: How well does the VFL (V) technique place fault statements at low ranks?

To answer RQ2, we compare the performance of the VFL (V) technique with that of the SFL technique using only the Top N entries of the generated ranked lists. The SFL and the VFL (V) columns indicate the number of faults found within Top 1, 5, 10, 20, and 50 lines.

In some cases, the number of faults found within the Top N is smaller than or equal to (white box) that of existing SFL techniques. Overall, however, the performance of the VFL (V) technique improves (gray box) dramatically in both Defects4j dataset and the Siemens suite in Table 7. For example, in Tarantula, the VFL (V) technique in Defects4j dataset localized 37 out of 224 faults in Top 1, while the SFL technique localized only 6 faults. In the Siemens suite, the SFL technique included 2 out of 120 faults in the Top 1, and the VFL (V) technique included 13 faults. Therefore, when the Tarantula similarity coefficient is applied, the VFL (V) technique includes 50 out of 344 faults in Top 1, while the SFL technique includes 8.

In addition, the SFL technique included 57, 85, 140, and 247 faults in the Top 5, Top 10, Top 20, and Top 50, and the VFL (V) techniques included 114, 157, 212, and 259 faults, respectively. The rates of increase of Top 1, Top 5, Top 10, Top 20, and Top 50 were 525%, 100%, 85%, 51% and 5%, respectively.

The accuracy depends on the applied similarity coefficient. However, the VFL (V) technique outperforms the SFL technique with all similarity coefficients. On average, the VFL (V) technique extends the number of programs included in the Top 1 from 11 to 44 (about 306%). In addition, programs included in the Top 5 were expanded by about 93% from 54 to 104. The program included in the Top 10 is expanded by about 89% from 77 to 145. The programs included in the Top 20 expanded by about 70% from 117 to 199. The programs included in the Top 50 expanded by about 24% from 203 to 251.

The minimum success criterion for finding faults is generally set from Top 1 to Top 50. However, most software developers only consider values up to the Top 5, even if a list of suspicious entities is obtained by applying the automated fault localization technique. In other words, if developers cannot find actual faults in the Top 5 of the suspicious entity ranked list, they find manual debugging ways [17]. Therefore, to help debugging tasks in the real world, faults must be found in the Top 5.
In experimental results, the existing SFL technique has only 54 programs that find the actual faults in the Top 5. Therefore, we propose the VFL (V) technique that can improve the localized faults from 54 to 104 (about 93%). In addition, the existing SFL technique has only 11 programs that find actual faults in the Top 1, but the VFL (V) technique included 44 programs with about 306% improvement in the Top 5. Based on the above experimental results, the VFL (V) technique is more helpful to the developer than the SFL technique.

As described above, the performance of the SFL technique is highly dependent on the similarity coefficient. If other similarity coefficients are applied to the same program, the results are different. Of course, this phenomenon appears in the VFL (V) technique as well because the basic concept of the VFL (V) technique is very similar to that of the SFL technique. Note that our results in Table 6 also show different results for each similarity coefficient.

When applying the SFL technique to the Defects4j dataset and Siemens suite, Tarantula has 8, Jaccard has 49, Naish, GP11, GP12, GP20 has 7, GP08 has 6, GP10 has 3 and GP26 has 4 programs in Top 1. Of course, the performance is different according to the similarity coefficient in the other Top N. Specifically, if Jaccard is applied to the Defects4j dataset, the accuracy of the technique is very high, being one of the outliers. Based on these results, we can see that applying Jaccard is the best way to localize faults in the Defects4j dataset. However, we additionally determined that it is not necessary to apply the VFL (V) technique to programs that already have a very high accuracy when applying the SFL technique. In other words, applying the VFL (V) technique to such programs is not necessarily better compared to the performance of the SFL technique. In Table 6, when Jaccard is applied to the Defects4j dataset, the SFL technique outperforms the VFL (V) technique (Top 5, 10, 20, 50). Since the SFL and the VFL (V) techniques include 47 and 59 programs in Top 1, respectively, 163 and 159 programs in Top 20, respectively, and 190 programs in Top 50, respectively.

This result ultimately requires further discussion, but it seems that this is caused by the ability of Jaccard to localize faults in the Defects4j dataset with high accuracy. In other words, we believe that it is difficult to improve the performance via the VFL (V) technique because the SFL technique using Jaccard is already overfitted for this benchmark. However, practically speaking it is rarely possible to know in advance the most optimal similarity coefficient for each benchmark [20, 35, 42, 44], and the VFL (V) technique outperforms the SFL technique for the remaining similarity coefficients except Jaccard.

In addition, when GP08, GP11, and GP20 are applied to the Siemens suite, the SFL technique outperforms the VFL (V) technique in the Top 50. However, developers do not check to the top 50 in practice, and applying the VFL (V) technique to these programs guarantees a better performance than the SFL technique. Therefore, we believe these outliers can be ignored. Although the SFL technique outperforms the VFL (V) technique for some programs, it is clear that the VFL (V) technique is an overall very effective technique, given the above reasons.

5.3. RQ3: Is the VFL (V) technique effective for every fault?

To answer RQ3, we compare the SFL and the VFL (V) ranks of actual fault locations by applying the SFL and the VFL (V) techniques to 344 programs. Fig. 4 indicates the results for 224 Defects4j dataset and 120 Siemens suite programs. The x-axis is the portion of programs, and the y-axis is similarity coefficients. Better defines the number of programs whose rank of faults found by the VFL (V) technique is lower than the rank of faults found by the SFL technique. Worse defines the number of programs whose rank of faults found by the VFL (V) technique is higher than the rank of faults found by the SFL technique. Finally, Same refers to the situation where the number of programs whose rank of faults found by the SFL technique and the number of programs whose rank of faults found by the VFL (V) technique are the same. N/A defines the number of programs that cannot be found by the VFL (V) technique.
For example, in Tarantula, the VFL (V) technique outperforms the SFL technique in 238 of 344 programs (about 70%). In addition, the accuracy is the same or worse in 15 (4%) and 62 (18%) of programs, respectively.

The accuracy depends on the similarity coefficient. However, the accuracy is improved in about 70% of programs for all similarity coefficients. The VFL (V) technique indicates a higher accuracy than the SFL technique in an average of 240 programs (about 70%). Also, the number of programs with the same and worse accuracy is 11 (3%) and 62 (18%), respectively.

The N/A of all the correlation coefficients is 29, which accounts for about 8% of the 344 programs. Therefore, the VFL (V) technique can be applied to more than 92% of the programs on average, and the accuracy is improved by more than 70%

5.4. RQA: Can we increase the VFL (V) technique performance and applicability by using additional techniques?

The SFL technique assigns a suspicious ratio to all statements covered by the test suite. Therefore, although the accuracy of this technique can be very low, it can always find faults. However, unlike the SFL technique, the VFL (V) technique only assigns suspicious ratios to variables. Therefore, it is necessary to handle statements that do not include any variable to improve the performance of the VFL (V) technique.

However, faults that cannot be found by the VFL (V) technique also cannot be accurately detected by the SFL technique. In Fig. 4, there are 29 N/A faults. This means that the location of the faults cannot be localized by the VFL (V) technique. Table 8 shows a complete list of programs that cannot find a fault using the VFL (V) technique. However, only 7 of the 29 programs (about 24%) can be included in the Top 10 with the existing SFL technique. In other words, the remaining 76% are programs that do not provide useful information to actual developers, even if the existing SFL technique is applied.

We found some common characteristics among the programs, after manual inspection of these 29 programs. That is, most programs invoke functions frequently. If there is a fault in the statements that invoke functions, it is difficult to find a fault with the VFL (V) technique. Therefore, we extend the VFL (V) technique to handle statements that do not contain variables by parsing the functions invocation information. This information can be used to find an additional 14 faults (Math 2–27, All Tcas faults in Table 8). In Fig. 4, there are 29 N/A faults with the VFL (V) technique. In Fig. 5, the number of N/A faults is reduced from 29 to 15 by applying the VFL (VF) technique. Among the 14 reduced programs, an average of 12 are included in the Better.

A comparison of the VFL (V) and the VFL (VF) techniques in Figs. 4 and 5 shows that the VFL (VF) technique improves the accuracy in more programs than the VFL (V) technique. The accuracy of the VFL (V) and the VFL (VF) techniques is improved by an average of 240 (about 70%) and 252 (about 73%) programs, respectively, compared to the SFL technique. In addition, to analyze the validity of the VFL (VF) technique, the evaluation metric is set to Top N and compared with the SFL and the VFL (V) techniques. The VFL (VF) columns in Table 7 indicate the number of programs included in the Top N when the VFL (VF) technique is applied. In the Top 1, Top 5, Top 10, Top 20, and Top 50, the accuracy of the VFL (VF) technique is improved by 28%, 23%, 11%, 14%, and 30%, respectively, compared with the VFL (V) technique. Also, the VFL (VF) technique extends the number of programs included in the Top 1 from 11 to 54, about 392%, compared with the SFL technique. In addition, programs included in the Top 5 are expanded by about 114% from 54 to 115. The programs included in the Top 10 are expanded by about 99% from 77 to 153. The programs included in the Top 20 are expanded by more than 92%.
about 80% from 117 to 211. The programs included in the Top 50 are expanded by about 31% from 203 to 266.

6. Discussion

6.1. Multiple faults

It is necessary to consider the number of faults when applying the proposed technique to real projects. However, it is difficult not only to determine the number of faults contained in this source code, but more generally to even determine whether the source code contains a single fault or multiple faults. Therefore, various studies dealing with multiple faults are already actively carried out, and we classify these researches into two types based on the treatment method.

The first is a sequential fault localization technique. Here, if some portion of test cases fail, the technique assumes that the program has one fault and then localizes the fault. Next, this technique fixes the fault and then runs the entire test again. (It is assumed that the found faults are completely fixed.) If another test case fails, this technique again assumes that there is an additional fault at another location and localizes this new fault. This technique repeatedly and sequentially localizes the faults until all test cases are passed.

The second is a parallel fault localization technique. If some portion of test cases fail, this technique first determines whether the fault is a single fault or multiple faults. If the source code contains multiple faults, this technique predicts the number of faults by using information such as coverage, model, and pattern. Then, the technique simultaneously localizes and fixes these multiple faults in parallel. Unlike sequential fault localization techniques, a parallel technique can be completed at once without repeating the fault localization task. However, if the test information is not sufficient, it is difficult to predict the number of faults. Therefore, in this case, the accuracy of such a parallel fault localization technique is not high [51–54].

In addition, since this multiple faults cannot be guaranteed to be mutually independent, an interference problem may occur between them. The sequential technique is to find faults one by one and fix them in any order. Therefore, the interference problem need not be considered. However, because the parallel fault localization technique finds all faults at once, this interference problem must be considered.

The interference problem means that more than two different faults that are combined affect each other. In general, interference consists of constructive, destructive, and constructive and destructive interference. If more than two different faults are existing at the same time, a new fault may be created that is different from the existing faults, and these faults are defined as constructive interference relations. On the contrary, if more than two different faults exist at the same time, all faults may be canceled out, and these faults are defined as destructive interference relations. Finally, if both constructive and destructive interference occur simultaneously, these faults are defined as constructive and destructive interference relations.

To apply the parallel VFL technique and verify this performance, multiple faults must be generated in combination of more than two faults. However, interference phenomenon of faults occurs in this process and it is very important to deal with them. In addition, this is a difficult problem that has been actively carried out but has not yet been resolved [58–62]. Actually, we have created a new program that contains multiple faults by combining several faults, and we have applied the parallel VFL technique here. As a result, some programs have been improved when parallel techniques were applied than when sequential techniques were applied. However, some of the program has the opposite result. In other words, the performance of parallel VFL technique is not stable and even highly dependent on interference between faults.

In addition, the VFL technique simply calculates the suspicious ratio by substituting the execution traces and test results information into the similarity coefficients like the SFL technique. Therefore, the performance of this technique is highly dependent on the similarity coefficients. Even the performances of sequential and parallel fault localization techniques are largely determined by the similarity coefficient. In other words, some similarity coefficients are more suitable for applying sequential techniques rather than parallel techniques, while for other similarity coefficients the opposite is true.

For the above reasons, we think that parallel VFL technique is not yet more effective than sequential VFL technique because there is not yet enough technique to help improve the parallel VFL technique by using the extracted variables and function invocations information. In other words, it is more effective to apply sequential fault localization technique rather than parallel to find multiple faults with the VFL technique. Therefore, to handle multiple faults with the VFL technique, we must apply this technique repeatedly to localize faults one at a time.

7. Threats to validity

7.1. External validity

The primary threat to external validity is that our results might not be generalizable. We evaluate performance by applying both Java programs (Defects4j dataset) and C programs (Siemens suite) to generalize our results. Defects4j dataset consists of four large-scale projects of 22,000 lines or more. Also, because it is an open source project, individual programs contain actual faults. Therefore, it is reasonable to evaluate the performance by applying the proposed technique to Defects4j dataset. In other words, this result can be generalized and is expected to show similar results to other projects.

However, the most widely used Siemens suite for evaluating the performance of FL techniques consists of very small toy programs of less than 1,000 lines. It is often pointed out that some researchers use an insufficient test suite. In other words, this project cannot represent all C programs. Likewise, the UNIX programs flex, grep, gzip, and make provided by SIR are large, but they also consist of programs that have artificial faults. Therefore, this project is also not realistic. In other words, the high performance of these projects does not guarantee high performance in all C programs. However, most of the studies performed in the past have been applied to this benchmark to evaluate performance. Although the Siemens suite might not be realistic, it is sufficient to identify the feasibility of the proposed technique, as in previous studies [39].

8. Related work

8.1. Spectrum-based fault localization techniques

The common feature of the SFL technique is that it gives suspicious ratio to statements based on execution trace and test result, and we classify them into three types according to research purpose. The first calculates the execution traces and test results of the program and then substitutes these values into the similarity coefficient formula. Although this technique is very traditional, it is still being studied because it has the advantage of very low computational cost [4].

Jones et al. [12] first designed Tarantula for software debugging. This technique provides a high suspicious ratio to entities that are close to the failed test case and far from the passed test case. Many similar techniques such as Jaccard [1] and Ochiai [32] have been designed. Dallmeier et al. [6] subsequently designed the AMPLE technique. This technique results in a suspicious ratio that is a function of the distance between covered failed and passed test cases. Compared to Tarantula, this technique assigns equally high suspicious to statements that are far from the failed test cases and close to passed. In other words, this technique considers only the difference between two test case sets, regardless of the test results. Naish et al. [23] suggested the Op and Op2 technique by experimentally analyzing various similarity coefficients. The Op2 technique is the representative in most SFL studies, and is often referred to as the Naish technique. Yoo [45] suggested GP 1–30 based
on a reformulation of various similarity coefficients with genetic programming techniques. Not all proposed similarity coefficients are optimal for all projects. However, the six GP similarity coefficients recommended by the authors showed high performance in most projects. Wong et al. [37,38] developed techniques for adding weights to failed test cases through various rules. Heuristic 1, 2, and 3 techniques empirically added different weights to the covered entities via the failed and the passed test cases. Also, Dstar additionally gave weights to the entities covered by the failed test case by squaring the numerator of the existing Kulczynski [38] similarity coefficient.

The second SFL technique type is to improve the performance of the fault localization technique by combining with other techniques or adding information to the suspicious ratio calculated by the traditional SFL technique described above. Mengshi et al. [56] suggested a technique that uses the PageRank technique to improve the performance of existing SFL techniques. This technique improves the accuracy of the SFL technique by further utilizing the connection relation (invocation frequency) between the functions and the information of functions covered by the test case. Divya Gopinath et al. [46] proposed a technique for improving the performance of the formula-based fault localization technique with SFL technique. This is a way to incrementally localize the faults while adding test cases one at a time based on the SFL technique. This technique computes the suspicious ratio of Tarantula and then selects the next test case most similar to this ratio to improve the efficiency of fault localization. Wong et al. [40,57] designed a technique to apply machine learning using the execution traces and test results of a program as training data. In addition, this technique utilizes the orthogonal matrix as test data. Therefore, the degrees to which individual statements affected the failed test case can be calculated and used as a suspicious ratio of individual statements.

The last SFL technique type is to compare and analyze various techniques. It is not easy to further improve the accuracy or efficiency of this technique, since this study has been very active up until recently. Therefore, many studies are being conducted to compare and analyze the features of existing similarity coefficients rather than researching improved technique. Xie et al. [42] theoretically analyzed the 30 existing similarity coefficients, and they grouped the similarity coefficients with similar characteristics into six groups. In addition, Tang et al. [35] compared and analyzed the accuracy of the coefficients using graph theory based on Xie’s results.

Xuan et al. [44] suggested a MULTRIC technique that combines multiple similarity coefficients. Because the performance of the similarity coefficient depends on the project, it is not possible to achieve the best performance using only one similarity coefficient. Therefore, the optimal situation is a combination of similarity coefficients based on historical results.

Lucia et al. [25] applied 40 similarity coefficients to the same project to compare the performance in the same environment. Experimental results have shown that there is no single best similarity coefficient for every project. It also proved that the performance of similarity coefficients depends on the number of faults (single or multiple).

Tien-Duy et al. [20] proposed Savant, which recommends one similarity coefficient that is most suitable for a project among several similarity coefficients. This technique uses various similarity coefficients to calculate rankings and learns their changes. Then, when the project is determined, it recommends the similarity coefficient that is most appropriate for the project.

9. Conclusion and future work

In this paper, we proposed a novel variable-based fault localization technique for more effective debugging. The existing SFL techniques assign suspicious ratios to individual statements based on the execution traces and test results. The proposed technique requires these two types of data as well as the name and location information of the variables and functions extracted from the faulty source code. This information can be easily extracted automatically without human effort using various existing open source tools. In addition, it is further utilized to solve the problems of existing SFL techniques that rely solely on coverage information. We have verified the significantly improved the accuracy of all correlation coefficients for 344 Java and C programs that include artificial and real faults.

Also, since the proposed technique is very lightweight, it is relatively easy to collaborate with other fault localization techniques. Although this requires the additional task of extracting variable and function information, it consumes very little time and cost with existing automation tools. In addition, if statements include multiple variables or function invocations, the VFL technique needs iterative calculations, but this is a simple computation, so it is lightweight enough to be ignored.

In addition, the existing SFL techniques calculate the suspicious ratios of all statements regardless of whether they contain variables and function invocations. On the other hand, the proposed technique first determines the suspicious variables and functions, and then computes only the suspicious ratios of the statements containing these variables and function invocations, thus reducing the overall computation.

To conclude, we expect that the proposed techniques are highly likely to evolve and will be very useful in the future. Furthermore, even though we are now assigning even weights without taking into account the type of statements, we will experiment in the future by assigning weights differently based on historical data. We will also consider assigning suspicious ratios only to statements containing some of the top-ranked variables in the calculated suspicious variable list.

Acknowledgments

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068179), and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2016R1D1A1B0394610).

References


