Received November 27, 2017, accepted January 8, 2018, date of publication January 23, 2018, date of current version March 13, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2796849

Spectrum-Based Fault Localization via Enlarging Non-Fault Region to Improve Fault Absolute Ranking

YONG WANG[1,2], ZHIQIU HUANG[1], BINGWU FANG[1], AND YONG LI[1]

[1]College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210000, China
[2]School of Computer and Information, Anhui Polytechnic University, Wuhu 241000, China

Corresponding author: Zhiqiu Huang (zqhuang@nuaa.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2016YFB1000802, in part by the National Natural Science Foundation of China under Grant 61772270 and Grant 61562087, in part by the Key Support Program Projects for Outstanding Young Talents of Anhui Province under Grant gxyqZD2016124, in part by the Anhui Natural Science Foundation under Grant KJ2016A252 and Grant 1608085MF147, and in part by the Natural Science Foundation of Jiangsu Province under Grant BK20170809.

ABSTRACT Spectrum-based fault localization (*SFL*) is a popular lightweight automatic software fault localization technique that uses coverage information of program execution to compute the likelihood of root cause of failure(s) for each program component and ranks them descending by their suspiciousness scores. However, some recent studies indicate an *SFL* technique to be useful only if the root cause(s) of failures is ranked at *top k*. Due to the nature of the *SFL* technique, it is impossible that the root fault(s) is always ranked at *top k*, which may interfere with the usefulness of *SFL* in practice. To solve this issue, an *SFL* technique via enlarging the non-fault region to further improve fault absolute ranking was proposed. The idea behind this is that we can intuitively improve fault absolute ranking for an *SFL* technique if some non-fault components ranked higher were excluded from the fault ranking list. In the approach, we enlarge the non-fault region iteratively to narrow down the suspicious region based on two scenarios, and then rank those components in the suspicious region using existing *SFL* techniques. The empirical results indicate that our approach significantly helps existing *SFL* techniques to further improve their usefulness.

INDEX TERMS Fault localization, absolute ranking, testing, debugging.

I. INTRODUCTION

Program debugging is time consuming, tedious and expensive. The effort for the detection (testing), locating and correction (fixing) of faults consumes between 30% and 90% of the development and maintenance budget of a project [1]. The use of dynamic coverage information collected during program testing, which contains useful information about fault localization, has been advocated to reduce debugging costs. Spectrum-based fault localization (*SFL*) is a such popular technique that uses the dynamic coverage information and testing results, measures the likelihood of a program component being a fault and ranks the components in descending order to help programmers identify root fault(s) [2]. Empirical studies show that *SFL* techniques can effectively guide programmers to examine suspicious codes and localize fault(s) [3]–[5]. However, the usefulness of these *SFL* techniques have not been thoroughly investigated [6].

Parnin and Orso [7] perform an empirical research on the usefulness of *SFL*. They find that several assumptions made by existing *SFL* works do not hold in practice. Recently, Xie *et al.* [8] revisit the usefulness of *SFL* via human focus-tracking analysis. They highlight that an inaccurate *SFL* result may lead to an even longer debugging process. Kochhar *et al.* [9] show that 98% of practitioners consider a fault localization technique to be useful only if it reports the root fault(s) within the *top 10* of the suspiciousness ranking. However, due to the nature of *SFL*, it does not always rank the root causes at top. Therefore, enhancement of absolute fault ranking for a *SFL* technique is of great significance to improve its usefulness in practice.

In a general fault localization process, programmers begin with a set of hypotheses, modify the hypotheses, select hypotheses for verification, and verify or refute the selected hypotheses until the fault is localized [10]. From a programmer's decision view point, all components can be

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

divided into three regions: suspicious region, neutral region and non-fault region [11]. Hence, a process of fault localization is essentially expressed as a series of two types of actions: enlarging the non-fault region and reducing the suspicious region via switching among the three regions.

However, in a *SFL* framework, all program components are assumed as hypothesis faults and are ranked based on their suspiciousness scores. Intuitively, some components ranked higher can be identified as non-fault components, and those components should be excluded from the suspicious region. For example, a non-fault component c_i in a buggy program is ranked higher in a fault ranking list outputted by a *SFL* tool. In the case, it is difficult to lower down the component's ranking based on general *SFL* techniques. Let us suppose the buggy program contained a single fault, and the failed testings satisfied with Fault&Failure Model(*RIP*) [12], which means the root fault must be reached in all failed runs, and cause software failures. Therefore, it is easily to see that c_i can be viewed as a non-fault if c_i is not reached in a failed execution. Therefore, we can use this information to exclude the component c_i from the suspicious region.

In this paper, we propose two novel *SFL* strategies to further improve the fault absolute ranking. The idea behind this is that we can intuitively improve fault absolute ranking for existing *SFLs* if some non-fault components ranked higher were excluded from the fault ranking list. In general, we first identify non-fault components based on different fault localization strategies iteratively, and then apply existing *SFL* techniques to generate a fault ranking list for all components in the suspicious region.

The remainder of this paper is organized as follows: Section II describes the background. Section III describes a motivation example. Section IV presents our technique. Section V presents our empirical research. Section VI presents a discussion. Section VII describes related works. Finally, Section VIII concludes with proposals for future work.

II. BACKGROUND

A. PRELIMINARY

1) BASIC CONCEPTS

Let $P = \{c_1, c_2, \ldots, c_m\}$ be a buggy program contained m components and $\Gamma = \{t_1, t_2, \ldots, t_n\}$ be a set of test cases for program P. Given an input d_i and expected output o_i for program P, a test case t_i is considered as a two-tuple $(d_i, o_i)(1 \leq i \leq n)$. The execution of P occurs upon inputting d_i outputs $o_i^{'}$. We say P passes the test case t_i if and only if $o_i == o_i^{'}$. Conversely, P fails on t_i. Therefore, Γ can be divided into two disjoint parts Γ_p and Γ_f. Γ_f and Γ_p be defined as follows:

$$\Gamma_p = \{t_i | o_i^{'} = P(d_i) \quad and \quad o_i^{'} = o_i\} \tag{1}$$

$$\Gamma_f = \{t_i | o_i^{'} = P(d_i) \quad and \quad o_i^{'} \neq o_i\} \tag{2}$$

During the process of software creation, a large amount of types of mistakes occur. To distinguish notations related with software faults, the IEEE conventions [13] are adopted here.

- Fault (bug): A static defect in a software.
- Failure: An external, incorrect behavior,which can be observed, with respect to the requirements or other description of the expected behavior.
- Error: An incorrect internal state, which cannot be observed directly, that is the manifestation of faults/bugs.

There are three necessary conditions for a failure to be observed, which are called the Fault&Failure Model. The model was proposed independently by DeMilli and Offutt [14] and Morell [15], and published as different notations. Current literature combines those notations as Reachability, Infection, and Propagation(*RIP model*) [12]. The means of three conditions are defined as follows:

- *R*eachability: The location(s) in the program that contain the fault must be reached.
- *I*nfection: The state of the program must be incorrect.
- *P*ropagation: The infected state must propagate to cause some output of the program to be incorrect.

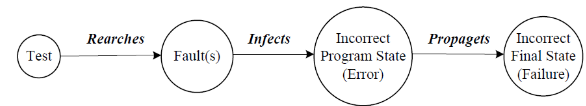


FIGURE 1. RIP model.

Fig. 1 illustrates the *RIP* model. To detect a fault, a test has to reach the location of the fault. This is illustrated in Fig. 1 where the test reaches the fault. The execution of the component in the faulty location must cause incorrect internal states, that is, the states must be infected, as illustrated by the narrow from fault to incorrect program states. For instance, a statement $result = a + b$ is calculate the sum of variable a and b to assign the variable *result*. However, a programmer makes a mistake and accidentally writes as $result = a - b$. Thus, during the buggy program running, the program state is different from the correct version after the program reaches at the buggy location. The incorrect internal program states then must propagate to an incorrect final sate possibly, which was observed by programmers.

Therefore, in a failed program spectrum, one of root fault(s) must be covered because a failure has been observed. We apply this simple information to design two different fault localization strategies: *Scenario 1* and *Scenario 2* in the section IV.

2) ASSUMPTIONS

Assumption 1: The output of a program P is deterministic.

Namely, a program P always produces the same result in different executions given a same input values.

Assumption 2: A failure occurred always satisfied with *RIP* model.

SFL techniques are characterised by measuring suspiciousness scores based on *coverage vector* of components and

their *Result vector*. Therefore, a *SFL* does not typically find a fault if the fault can not be reached in program executions, such as *Missing code* faults, *variable declaration* faults and et al., which have no faults be covered in program executions, although those cases can still be interpreted by *RIP* model.

Assumption 3: There exists a test oracle that determines the status of a test execution for program *P*, i.e., "successful" or "failed".

Constructing the test oracle is a difficult, independent research problem. Assumption 3 is made to simplify the approach. Given a test oracle, our approach may be fully automated to collect *Coverage Matrix* and *Result Vector*.

Assumption 4: There exist many failed test cases, and we can collect many different program spectra by executing those test cases.

Our approach requires many failed program spectra to rule out non-fault components iteratively. Our approach cannot improve a *SFL* technique if there is only one failed program spectrum.

B. SPECTRUM-BASED FAULT LOCALIZATION

SFL techniques calculate the likelihood of a program component being a fault for each component in the program [3]. They exploit the *program spectrum*, which includes information about which component was covered in each execution, from failed and successful program executions. Collecting program spectra using an instrumented program version is an lightweight analysis method, and a *program spectrum* provides a dynamic view of the behavior of the system under test [16]. As a *SFL* technique only registers whether a component is covered during a certain execution, binary flags('0'/'1') can be used for each component, and the program spectra is also called a hit spectra [16].

The hit spectra is mapped as *Coverage Matrix A*, which is a binary $N \times M$ binary matrix, where N indicates the number of executions, and M indicates the number of instrumented program components of the buggy program. The result of each execution can be viewed as a N-length *error detection vector*, also called *Result Vector e*, where '0' indicates success and '1' indicates failure. The (A, e) works as an input for a *SFL* approach. The *Coverage Matrix* and *Result vector* are shown in Fig. 2 and are used to compute the suspiciousness score for each component based on a certain suspiciousness metric. All program components are then ranked in descending order based on their suspiciousness scores for programmers to manually check.

$$N \text{ spectra} \quad \overbrace{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{M2} & \cdots & a_{NM} \end{bmatrix}}^{M \text{ components}} \quad \overbrace{\begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix}}^{\text{error detection}}$$

FIGURE 2. Program Spectrum and Error Detection Vector.

The key for a *SFL* is the similarity metric used to compute the suspiciousness scores. There exist several similarity metrics that can be used to compute suspiciousness scores [17]. Table 1 shows four suspiciousness metrics of well-known fault localization techniques: *Jaccard* [4], *Tarantula* [2], *Ochiai* [4], and *Dstar* [29]. Given a *program spectrum* and *Result Vector*, we calculate n_{00}, n_{01}, n_{10}, and n_{11}. For each program component j, $n_{00}(j)$ is the number of successful executions in which component j is not covered, $n_{01}(j)$ is the number of failed executions in which component j is not covered, $n_{10}(j)$ is the number of successful execution in which component j is covered, and $n_{11}(j)$ is the number of failed executions in which component j is covered. Four well-known suspiciousness metrics for *SFL* are defined in Table 1.

TABLE 1. Four well-known suspiciousness metrics for ***SFL***.

Name	Metric
Jaccard	$\frac{n_{11}(s)}{n_{01}(s)+n_{10}(s)+n_{11}(s)}$
Tarantula	$\frac{(n_{10}(s)+n_{00}(s))\times n_{11}(s)}{(n_{01}(s)+n_{11}(s))\times n_{10}(s)+(n_{10}(s)+n_{00}(s))\times n_{11}(s)}$
Ochiai	$\frac{n_{11}(s)}{\sqrt{(n_{01}(s)+n_{11}(s))\times(n_{11}(s)+n_{10}(s))}}$
Dstar	$\frac{(n_{11}(s))^{*}}{n_{01}(s)+n_{10}(s)}$

C. FAULT LOCALIZATION DECISION VIEW

Fig. 3 depicts a general fault localization decision view for programmers [11]. All program components *CSet* were classified into three subsets.

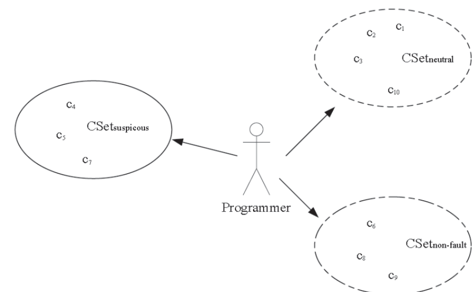


FIGURE 3. Fault localization decision model.

Definition 1: $CSet_{suspicious}$ is the suspicious set, which describes a suspicious region that might contain root fault(s).

Definition 2: $CSet_{non-fault}$ is the non-fault set. $CSet_{non-fault}$ describes a trusted region in which the components appear to have no-faults.

Definition 3: $CSet_{neutral}$ is the neutral set. $CSet_{neutral}$ describes a neutral region in which it is uncertain whether the components contain faults.

At the beginning of fault localization, the programmer does not know any of the components, and the initial state of each subset is as follows:

- $CSet_{suspicious} = \emptyset$
- $CSet_{neutral} = \{c_1, c_2, \ldots, c_m\}$
- $CSet_{non-fault} = \emptyset$

The process of fault localization can be viewed as a series of *movement actions* among the three regions. At the final

stage of the fault localization process, the fault component c_i has been located for a buggy program contained a single fault, and the contents of the three subsets are as follows:

- $CSet_{suspicious} = \{c_i\}$
- $CSet_{neutral} = \emptyset$
- $CSet_{non-fault} = \{c_1, c_2, \ldots, c_m\}/\{c_i\}$

In general, these *movement actions* are performed in the programmer's mind, and the actual process behavior cannot be observed externally. To simplify the decision model, our approach expresses the *movement actions* as a sequence of two types of action: enlarging the non-fault region and narrowing the suspicious region.

III. MOTIVATION EXAMPLE

We begin with an example shown in Fig. 4 to illustrate our motivation. Considering program *Example* in Fig. 4, which has a bug at *block 2*. The *Example* should calculate the sum of variables a and b but instead is accidentally written as $result = a - b$. Columns 1 to 2 show the block numbers associated with the *Example* statements. Column 3 through 14 represent $t_1 - t_{12}$, respectively. The program spectrum of component c are represented in the columns and indicate whether the corresponding block is involved in the testing ('0' for involved, '1' for not involved). The successful/failed status of the tests are given in the bottom row, where '1' indicates a failure and '0' indicates a success. The columns representing failed test cases are highlighted. The *Dstar* tool is used to measure the suspiciousness of a *component c* being a fault. Each *component c* is assigned a normalized suspiciousness score between 0 and 1.

No.BB	Prog1(a,b)	Coverage (0=executed,1=unexecuted) t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	Normalized DStar Scores
1	if(a<10&&b<10)	1	1	1	1	1	1	1	1	1	1	1	1	0.22
2	result=a-b;//bug if(result>0)	1	1	1	1	0	1	1	1	1	1	1	1	0.25
3	print("positive");	1	0	0	0	0	1	1	0	0	1	1	1	0.01
4	else if(result==0)	0	1	1	1	0	0	0	1	1	0	0	0	0.33
5	print("zero");	0	1	0	0	0	0	0	0	1	0	0	0	0.03
6	else print("negative")	0	0	1	1	0	0	0	1	0	0	0	0	0.15
7	else print("invalid input");	0	0	0	0	1	0	0	0	0	0	0	0	0.00
Program Results (0=successful,1=failed)		1	1	1	1	0	0	0	0	0	0	0	0	

FIGURE 4. An example contained single fault.

As mentioned in the above section, *SFL* techniques, such as *Dstar*, focus on computing independently suspiciousness scores for each component according to the selected suspiciousness metric, which measures the correlation between a component and failures, and ranks them. In our example, c_4 has the highest score of 0.33 among the seven *components*, which is higher than root fault c_2. In this case, it is difficult to lower ranking for c_4 or improve ranking for c_2 based on suspiciousness metric of general *SFL*. Based on programmers decision model, the three subsets were generated as $CSet_{suspicious} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, $CSet_{non-fault} = \{c_7\}$, $CSet_{neutral} = \emptyset$ based on the *Dstar* tool.

Due to the *example* contained single-fault, the root fault must be reached in all failed program spectra based on *RIP* model. In other words, if a component is not covered by a failed program spectrum, then the component is non-fault.

For a failed program spectrum collected by running t_1, we can infer c_4, c_5, c_6, and c_7 belong to the non-fault region. Hence, for each failed program spectrum, we can infer which components are non-fault components. Finally, we obtain $CSet_{suspicous} = \{c_1, c_2\}$, $CSet_{non-fault} = \{c_3, c_4, c_5, c_6, c_7\}$. For the subset $CSet_{suspicious}$, we use a *SFL* technique, such as *Dstar*, to compute suspiciousness scores and rank them. The result is that the root fault c_2 is ranked at top.

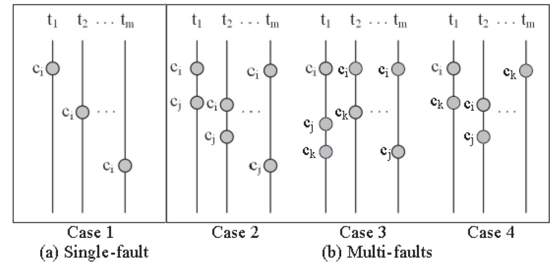


FIGURE 5. Four cases of fault triggering based on RIP model.

IV. OUR APPROACH

A. THE MAIN IDEA

The goal of our approach is to further improve fault absolute ranking via enlarging non-fault region. Therefore, it is crucial that how to identify which component is non-fault. According *RIP model*, there exist four fault triggering schemas, which as shown Fig. 5:

- CASE 1: The single root fault must be contained in all failed spectra for single-fault programs;
- CASE 2: All root faults be contained in all failed spectra due to the dynamic control flows during executing failed test cases;
- CASE 3: Some root faults be contained in all failed spectra, and others not;
- CASE 4: All root faults are not contained in all failed spectra, but contained in some failed spectra.

Let us closely look the four cases, we can combined the four cases into two scenarios. The CASE 1 and CASE 2 can all be viewed as *scenario 1*, and the CASE 4 can be viewed as *scenario 2*. For CASE 3, we divide two groups: one belong to *scenario 1* and others belong to *scenario 2*. The *scenario 1* can be defined as all root fault(s) be contained in all failed spectra; The *scenario 2* can be defined as all root faults are not contained in all failed spectra,but contained in some failed spectra. Therefore, we can narrow down $CSet_{suspicious}$ for a buggy program if we know which *scenario* is belonged to.

After $CSet_{suspicious}$ is obtained, the suspiciousness metric ρ_b is defined such that the higher the suspiciousness score of program component c is, the more likely c being a fault. In the field of fault localization, several suspiciousness metrics ρ_b exist. We select *Dstar* as our benchmark metric to compute the suspiciousness for each *component*. *Dstar* was proposed by Wong et al. in 2014, and it is more effective than others

SFL techniques based on their experiments [29]. The formula of *Dstar* is as follows:

$$\rho_b^{Dstar}(e) = \frac{(n_{11}(e))^*}{n_{01}(e) + n_{10}(e)} \quad (3)$$

In formula 3, $*$ is a power of n_{11} whose value is greater than or equal to 1. The relationship between the effectiveness of *Dstar* and the value of $*$ is examined empirically, and it is discovered that the effectiveness increases alone with $*$ before it reaches a critical value [29]. In our experimental study, we set $*$ equal to 2.

B. STRATEGY FOR SCENARIO 1

In *scenario 1*, we easily know that those components that do not involve once a failed program spectrum are non-fault components. Let program P be $\{c_1, c_2, \ldots, c_m\}$, and its test cases $\Gamma = \{t_1, .., t_n\}$. A *Coverage Matrix A* is collected by executing Γ. For a failed t_i, the set of non-fault components can be defined as Formula 4.

$$cset_{non-fault}(A, t_i) = \{c_j | 0 \le j \le m, A_{ij} = 0\} \quad (4)$$

Taking the motivation *example* as an illustration, t_3 is a failed test case, and it is easy to infer that c_3, c_5, and c_7 are non-fault components; that is to say, $cset_{non-fault}(A, t_3) = \{c_3, c_5, c_7\}$. According to *Assumption 4*, we have many failed test cases that produce different program spectra for program P. Therefore, we can enlarge the range of non-fault components iteratively. The set of non-fault components can be defined as Formula 5.

$$CSet_{non-fault}(A, \Gamma_f) = \bigcup_{t_i \in \Gamma_f} cset_{non-fault}(A, t_i) \quad (5)$$

In the initial phase, $CSet = \{c_1, c_2, \ldots, c_n\}$, $CSet_{suspicious} = \emptyset$, and $CSet_{non-fault} = \emptyset$. For each failed spectrum i, we enlarge the region of non-fault components iteratively. Finally, $CSet_{suspicious}$ can be computed as Formula 6.

$$CSet_{suspicious}(A, \Gamma_f) = CSet \setminus CSet_{non-fault}(A, \Gamma_f) \quad (6)$$

C. STRATEGY FOR SCENARIO 2

In *scenario 2*, all faulty components are covered in some failed spectra, and not covered in all failed spectrum. Therefore, there exist two types of components can be identified as non-fault components:

- (1) Components are contained in all failed executions.
- (2) Components are never contained in all failed executions.

For a failed t_i, all covered components can be viewed provisionally as suspicious components, and the set of suspicious components can be defined as Formula 7. Similarly, components that are not executed can be viewed provisionally as neutral components, and the set of neutral components be defined as Formula 8.

$$cset_{suspicious}(A, t_i) = \{c_j | 0 \le j \le m, A_{ij} = 1\} \quad (7)$$

$$cset_{neural}(A, t_i) = \{c_j | 0 \le j \le m, A_{ij} = 0\} \quad (8)$$

For each failed test case t_i, we compute $cset_{suspicious}(A, t_i)$ and $cset_{neural}(A, t_i)$. For all failed test cases Γ_f, we identify the two types of non-fault components as mentioned above. For type (1), the set of non-fault components can be defined as Formula 9. For type (2), the set of non-fault components can be defined as Formula 10.

$$CSet1_{non-fault}(A, \Gamma_f) = \bigcap_{t_i \in \Gamma_f} cset_{suspicious}(A, t_i) \quad (9)$$

$$CSet2_{non-fault}(A, \Gamma_f) = \bigcap_{t_i \in \Gamma_f} cset_{neural}(A, t_i) \quad (10)$$

In the initial phase, $CSet = \{c_1, c_2, \ldots, c_n\}$, $CSet_{suspicious} = \emptyset$, and $CSet_{non-fault} = \emptyset$. For each failed program spectrum i, we can compute $cset_{suspicious}(A, t_i)$ and $cset_{neural}(A, t_i)$. Then, we compute $CSet1_{non-fault}(A, \Gamma_f)$ and $CSet2_{non-fault}(A, \Gamma_f)$. Finally, $CSet_{non-fault}$ is computed as Formula 11, and $CSet_{suspicious}$ is computed as Formula 12.

$$CSet_{non-fault} = CSet1_{non-fault} \cup CSet2_{non-fault} \quad (11)$$

$$CSet_{suspicious} = CSet \setminus CS_{non-fault} \quad (12)$$

V. EXPERIMENT

We built a fault localization prototype tool called SFL_{ernc}, which represents *SFL* via ***E**nlarging the **R**egion of **N**on-fault **C**omponents*, to implement our approach, and we present an empirical evaluation of SFL_{ernc}. To evaluate the effectiveness of our approach, we run the *SFL* technique *Dstar* as a benchmark. In particular, we search for answers to the following three research questions:

- **RQ1** How prevalent the four fault triggering schemas are in buggy programs?
- **RQ2** What is the effectiveness of SFL_{ernc} on *Scenario 1*?
- **RQ3** What is the effectiveness of SFL_{ernc} on *Scenario 2*?

TABLE 2. Subjects used for empirical studies.

Program	LOC	BBs	Vers	Testcases	Description
print_token	478	122	5	4130	Lexical analyzer
print_token2	399	135	9	4115	Lexical analyzer
replace	512	153	26	5542	Pattern matcher
schedule	292	73	8	2650	Priority scheduler
schedule2	301	77	8	2710	Priority scheduler
tcas	141	23	36	1608	Aircraft control
tot_inf	440	74	19	1052	Info measure

A. EXPERIMENTAL DESIGN

1) SUBJECT PROGRAMS

For our studies, we used 7 C programs from SIR [18], which contain tcas, tokens, tokens2, replace, tot_inf, schedule, and schedule2, as experiment objects. Table 2 shows the detailed characteristics of these programs. To meet *Assumption 3*, we created a fault-free version of each program and constructed a test oracle to determine the status of a test execution for each program version. Columns 2 to 3 describe the

number of uncommented code lines and basic blocks in these programs, the 4th column shows the number of buggy program versions, the 5th column shows the number of test cases for each program version, and the last column gives a detailed description for each program. These programs have been used in many fault localization studies. We check those program versions, and there does not exist *Missing code* faults in the program versions. Each subject program version contains a single fault or multi-faults. There are a total of 132 versions in the subject programs. We exclude 20 versions that are seeded by bugs residing in variable declarations or that have no failed tests. In total, our experiments include 112 faulty versions of the Siemens test suite from SIR [18].

2) IMPROVEMENT METRIC

Effectiveness metrics are an important way to perform accurate and objective comparisons. There are two main metrics: *PDG-based metric,Renieres2003*, and *Ranking-based metric,Wong2014*, which used to measure quality of fault localization in the field.

The two metrics for *SFLs* normalize the fault ranking with respect to the size of the program. For instance, the root cause of program *P* with *1200 components* at ranking *52*, when expressed as a percentage, suggests that only 4.33% of the components should be checked. This result, at first sight, may be fairly positive. However, in practice, this result would not help programmers in the debugging process [7], [8], [19]. Our goal is to further improve the absolute ranking. Therefore, these two metrics cannot be used directly to evaluate our approach. To measure the effectiveness of our approach, we focus on improving the absolute ranking for suspicious program components in the *top k*. Wang *et al.* [27] propose a *improvement metric* to measure the absolute ranking improvement. Let *B* be the absolute ranking of a root cause generated by an *SFL* tool for a buggy version and *A* be the absolute ranking of the root cause generated by our approach for the same version. The improvement metric comparing the *SFL* technique with our approach can be defined as Formula 13:

$$Improvement_{SFL_{ernc}}^{SFL}(A, B) = \frac{B - A}{B} \times 100\% \qquad (13)$$

In this metric, we assume that a absolute ranking is improved 100% by our approach if the root fault was ranked at top improved from *top k*. For the purpose, we assign the absolute ranking's index starting from $0(\frac{k-0}{k} \times 100\% = 100\%)$. Based on the *improvement metric*, if a *SFL* ranked a root fault at 20 for a buggy program and our approach ranked it at 17, we can say our improvement is $\frac{19-16}{19} \times 100\% = 15.8\%$.

B. RESULTS

1) RQ1:HOW PREVALENT THE FAULT TRIGGERING SCHEMAS IN PROGRAMS?

We are interested that how prevalent the four fault triggering schemas in our experiment subjects, which are widely used in fault localization community. We firstly divide the *Siemens programs* into two groups: single-fault programs and multi-fault programs. The fault triggering schema of single-fault programs is obviously *CASE 1*. There exist 100 single-fault program versions and 12 multi-fault program versions. Due to most of *Siemens programs* are single-fault programs, therefore, the distribution does not answer this question. Recently, Perez et al. conducted an experiment to study prevalence of single-fault fixes in real programs. They have found 1375 fixes on over 70 projects, and out of all fixes,82.5% were single-faulted [24]. Therefore, *CASE 1* should be dominant in practice.

Yet, we are still interested that how prevalent the other three fault triggering schemas in multi-fault programs. Due to there exist fewer multi-fault programs in our experiment subjects, we have extended the subject programs with program versions where we can activate arbitrary combinations of multiple faults in *Siemens programs*. For this purpose, we limit ourselves to a selection of 83 out of the 105 faults, based on a criteria such as faults being attributable to a single line of code, to enable unambiguous evaluation.

TABLE 3. Additional program versions for *Scenario 2*.

Program	C=2	C=3	C=4
print_token	8	6	1
print_token2	20	33	9
replace	32	30	30
schedule	20	20	11
schedule2	18	35	40
tcas	30	30	30
tot_inf	38	32	32

We randomly select some combinations of the faults for each program, and the detailed information is shown in Table 3. In Table 3, *C* represents the number of injected faults (cardinality). Columns 2 to 4 describe the number of program versions which are randomly injected with *C* faults.

Fig. 6, 7, 8 depicts the three fault triggering schema distributions in those programs. The results show that the *CASE 3* is dominate among the three *CASEs*, and the proportion of *CASE 2* and *CASE 4* increase gradually with the number of faults increasing.

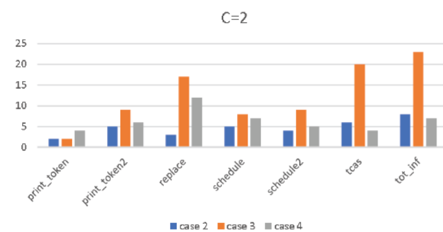


FIGURE 6. Three CASEs distributions for 2 fault programs.

2) RQ2: *SCENARIO 1*

In this section, we compare our approach with *Dstar* in *Scenario 1*. We first investigate 112 *Siemens programs*,

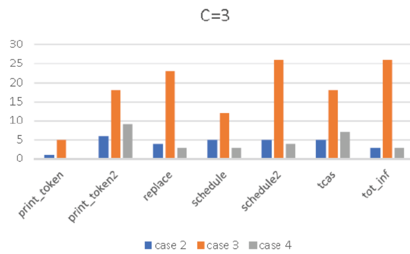


FIGURE 7. Three CASEs distributions for 3 fault programs.

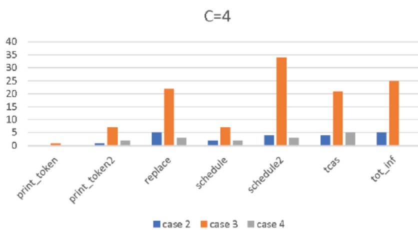


FIGURE 8. Three CASEs distributions for 4 fault programs.

then select 105 programs belong to *Scenario 1* which contain 100 single-fault programs and 5 multi-faults programs. We run *Dstar* and our approach independently and collect the two fault ranking lists.

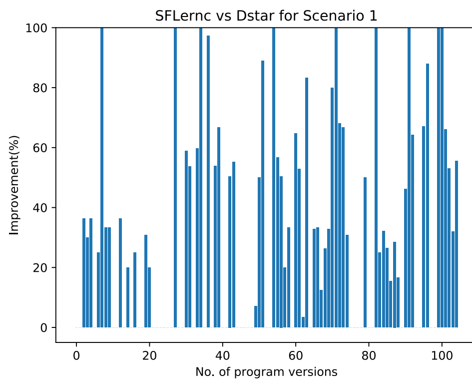


FIGURE 9. Comparing Effectiveness for *Scenario 1*.

We use the *improvement metric* mentioned above to estimate the improvement by our approach. We calculate the *improvement* based on the two fault rank lists generated by our approach and *Dstar*. Fig. 9 shows the results of the study. In the 105 fault versions, our approach performs better than *Dstar* tool on 63 versions, and 42 program versions have no improvement that includes 14 program versions ranked at top based on *Dstar* tool, which have no room to improve. The results show that no program versions perform worse than *Dstar*. In those 63 versions, the average improvement is 52.1%, and it is worth mentioning that the root fault of 9 buggy programs are ranked at top in the fault ranking list.

3) RQ3: *SCENARIO 2*

To answer this question, we firstly use 7 multi-fault programs belonged to *Scenario 2* in the 112 *Siemens programs* to evaluate the effectiveness of our approach SFL_{ernc}. Table 4 presents the *improvement* in absolute ranking for the 7 programs. For example, for *replacev21* that contained 3 faults, our approach ranks the root cause of failures at *top 8*, while *Dstar* ranks the root cause of failures at *top 37*. Therefore, based on the *improvement metric*, the improvement is 78.3%.

TABLE 4. Comparing Effectiveness of SFL_{ernc} with Dstar for *Scenario2*.

Program	# bugs	R_{Dstar}	$R_{SFL_{ernc}}$	improvement
tcasv10	2	4	2	50.0%
tcasv11	3	4	2	50.0%
tcasv31	3	1	0	100%
tcasv32	3	2	0	100%
tcasv40	2	9	0	100%
schedule2v7	4	0	0	0.0%
replacev21	3	37	8	78.3%

From Table 4, we observe that our approach effectively improves the absolute ranking compared with *Dstar*, except for program *schedule2v7*. One of the root faults for *schedule2v7* have already ranked at top based on *Dstar*, hence, there was no room to improve. It is worth mentioning that 3 program root faults were ranked at top.

Due to there exist few of multi-fault programs in *Siemens programs*, we use the extend program versions mentioned above section. There exist total 505 multi-fault program versions, we select 89 programs belong entirely to *Scenario 2*. We use the *improvement metric* mentioned above to estimate the improvement by our approach. We calculate the *improvement* based on the two fault ranking lists generated by our approach and *Dstar*.

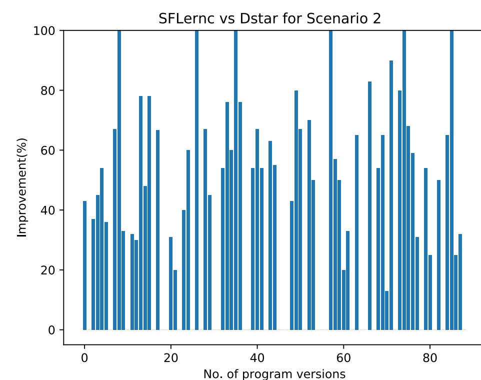


FIGURE 10. Comparing Effectiveness for *Scenario 2*.

Fig. 10 shows the result of the study. The result shows that there exist 59 versions perform better than *Dstar* (*66.3%*), and 30 versions have no improvement (*33.7%*). There exist 13 program versions whose root faults are ranked at top based on *Dstar* in the 30 program versions, which have no room to improve. In the 59 improved program versions, the average improvement is 67.7%. Therefore, the result reconfirm our approach for *Scenario 2* can further improve improve absolute ranking for existing *SFL* techniques.

VI. DISCUSSION

Based on our approach, we combined with a *SFL* would lead to the best performance for single-fault programs. The reason

for our *scenario 1* strategy is superiority for single-fault programs is established in terms of the following theorem.

Theorem 1: For single-fault programs, given the available set of observation (A, e), the fault ranking generated by *scenario 1* strategy is theoretically optimal.

Proof 1: For single-fault programs, the root fault *has* to be covered in all failed runs. If a component which did not involved in all failed runs can not explain all failed runs, the component is viewed as non-fault component. This implies that for the remaining components $n_{11}(e)$ is equal to the number of failed runs, and $n_{01}(e)$ is equal to *0*. Therefore, the constant $n_{11}(e)$ can be ignored concerning the *ranking*. The suspiciousness metric of *Dstar* can be written as:

$$\rho_b^{Dstar}(e) = \frac{1}{n_{10}(e)} \tag{14}$$

As we have already ruled out all components whose $n_{11} < n_f$, the rank list outputted by SFL_{ernc} is optimal.

Although single-fault programs are dominate in practice [24], there still exist some multi-fault programs. More importantly, we don't know whether a program is a single fault program or a multi-fault program, much less one of which is fault triggering schema belonged to. Therefore, we can not apply the two scenario fault localization strategies directly. Let us look closer at the two strategies, the two suspicious component sets outputted by SFL_{ernc} are disjoined. We could combine the two fault localization strategies to locate fault(s).

However, how to combine the two strategies to perform fault localization? Recently, Kochhar et al. show that 98% of practitioners consider a fault localization technique to be useful only if it reports the root fault(s) within the *top 10* of the suspiciousness ranking [9]. We agree with their opinions. Therefore, there exist many heuristic methods to combine the two strategies, such as we could first check *top 5* outputted by *scenario 1*, if we can not find the root fault, then we find next *top 5* outputted by *scenario 2*.

Considering single-fault programs are dominate in practice and our experiment results, *scenario 1* is prevalent. Therefore, we recommend that programmers can check *top 10* outputted by *scenario 1* firstly. If they can not find root cause of failures, then they try to check the *top 5* outputted by *scenario 2*. If they cannot find the root cause of failures in the two fault ranking list, we suggest programmers switch to other fault localization techniques to perform debugging.

In a word, it is very meaningful to perform debugging if the root cause of failures is ranked at *top* in any the two rankings. Additionally, if we find a root cause of failures in the ranking generated by *scenario 2*, we can know there exist multi-faults in the buggy program, and it is very important to guide programmers to perform next task.

VII. RELATED WORKS

Over the past decade, many studies have been performed on *SFL*. Various methods have been used to measure the likelihood of faults for program components, such as *Tarantula,Jones2005*, *Ochiai,Abreu2009*, *Dstar,Wong2014*, *SOBER,Liu2005* and many others (e,g, [20], [28]). Their experimental results show that *SFL* can help programmers to localize faults [5].

However, some recent studies indicate a *SFL* to be useful only if the root cause(s) of failures is ranked at (top k) [6], [9], [19], [23]. Particularly, Pearson et al. highlight that fault localization techniques performance on artificial faults is not predictive for real faults, and all *SFL* techniques are equally good [23]. To improve the usefulness of *SFL*, several *SFL* techniques have been combined with program analysis, which could also be used to further improve the absolute ranking. Zhang *et al.* [31] used edge profiles to represent successful executions and failed executions and contrasted them to model how each basic block contributes to failures by abstractly propagating the infected program states to their adjacent basic blocks through control flow edges. Wang *et al.* [27] propose an lightweight fault localization combined with fault-context to improve fault absolute ranking. Naish *et al.* [22] highlighted that failed test cases that cover fewer statements should be given more weight and have more influence on the fault rank and associated varying *weights* with the failed test cases. Santelices *et al.* [26] proposed an *SFL* using multiple coverage types. Different those works, our approach rule out the interfere with non-fault components, and those *SFL* approaches can be combined with our approach to further improve the absolute ranking.

There also exist some works are similar as our approach. Xie et al. proposed a refinement method to improve the accuracy of *SFL* [30]. In their approach, all the components are divided into an unsuspicious group and a suspicious group. They believe that suspicious group contains statements which have shown up at least once in a failed spectrum, while in the unsuspicious group, no statement has shown up in any failed test run. In our approach, the non-fault region is defined based on two fault triggering schemas. Although they categorized all components into two groups which is similar to our approach, their approach is different from ours. Our approach is also different from Set-union and Set-intersection, which are two heuristic models proposed by Renieres and Reiss [25]. They both use a single failed spectrum and all successful program spectra. Set-union focuses on the program component that is executed by the failed test but not by any of the successful tests, and Set-intersection excludes the component that is executed by all successful tests but not by the failed test.

VIII. CONCLUSION

In this paper, we propose a fault localization technique named SFL_{ernc} to automatically enlarge non-fault region to perform fault localization. According to four fault triggering schemas, we divide them into two fault localization strategies. By evaluating 112 programs and extended subjects, SFL_{ern} was shown to effectively improve the absolute ranking compared with *SFL* techniques in certain scenario.

In our future work, we plan to perform more empirical studies to further evaluate the effectiveness of

our approach. Moreover, this work is part of a large effort to develop automatic debugging approaches. The next step in our project is to build the relationship for suspicious components and non-fault components to assist the programmer to perform fault localization and to apply our technique to program slice spectra to expand the range of non-fault components.

REFERENCES

[1] B. Beizer, *Software Testing Techniques*. New Delhi, India: Dreamtech Press, 2003.

[2] J. A. Jones and M. J. Harrold, ''Empirical evaluation of the tarantula automatic fault-localization technique,'' in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Nov. 2005, pp. 273–282.

[3] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, ''On the accuracy of spectrum-based fault localization,'' in *Proc. Testing, Acad. Ind. Conf. Pract. Res. Techn.-MUTATION TAICPART-MUTATION*, 2007, pp. 89–98.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, ''A practical evaluation of spectrum-based fault localization,'' *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009.

[5] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, ''A survey on software fault localization,'' *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[6] X. Xia, L. Bao, D. Lo, and S. Li, '''Automated debugging considered harmful' considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems,'' in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, Oct. 2016, pp. 267–278.

[7] C. Parnin and A. Orso, ''Are automated debugging techniques actually helping programmers?'' in *Proc. Int. Symp. Softw. Test. Anal.*, 2011, pp. 199–209.

[8] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, ''Revisit of automatic debugging via human focus-tracking analysis,'' in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, May 2016, pp. 808–819.

[9] P. S. Kochhar, X. Xia, D. Lo, and S. Li, ''Practitioners' expectations on automated fault localization,'' in *Proc. 25th Int. Symp. Softw. Test. Anal.*, 2016, pp. 165–176.

[10] K. Araki, Z. Furukawa, and J. Cheng, ''A general framework for debugging,'' *IEEE Softw.*, vol. 8, no. 3, pp. 14–20, May 1991.

[11] S. Uchida *et al.*, ''A multiple-view analysis model of debugging processes,'' in *Proc. Int. Symp. Empirical Softw. Eng.*, 2002, pp. 139–147.

[12] P. Ammann and A. J. Offutt, *Introduction to Software Testing*. Cambridge, U.K: Cambridge Univ. Press, 2016.

[13] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr, ''Basic concepts and taxonomy of dependable and secure computing,'' *IEEE Trans. Depend. Sec. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[14] R. A. DeMilli and A. J. Offutt, ''Constraint-based automatic test data generation,'' *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.

[15] L. J. Morell, ''A theory of error-based testing,'' Dept. Comput. Sci., Maryland Univ., College Park, MD, USA, Tech. Rep.TR-1395, 1984.

[16] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, ''An empirical investigation of the relationship between spectra differences and regression faults,'' *Softw. Test. Verification Rel.*, vol. 10, no. 3, pp. 171–194, 2000.

[17] S. S. Choi, S. H. Cha, and C. C. Tappert, ''A survey of binary similarity and distance measures,'' *J. Syst., Informat.*, vol. 8, no. 1, pp. 43–48, 2010.

[18] H. Do, S. Elbaum, and G. Rothermel, ''Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact,'' *Empirical Softw. Eng.*, vol. 10, no. 4, pp. 405–435, 2005.

[19] T. D. B. Le, D. Lo, and F. Thung, ''Should I follow this fault localization tool's output?'' *Empirical Softw. Eng.*, vol. 20, no. 5, pp. 1237–1274, 2015.

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, ''Scalable statistical bug isolation,'' *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[21] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, ''SOBER: Statistical model-based bug localization,'' *ACM SIGSOFT Softw. Eng. Notes. ACM*, vol. 30, no. 5, pp. 286–295, 2005.

[22] L. Naish, H. J. Lee, and K. Ramamohanarao, ''Spectral debugging with weights and incremental ranking,'' in *Proc. Asia–Pacific Softw. Eng. Conf. (APSEC)*, 2009, pp. 168–175.

[23] S. Pearson *et al.*, ''Evaluating and improving fault localization,'' in *Proc. 39th Int. Conf. Softw. Eng.*, May 2017, pp. 609–620.

[24] A. Perez, R. Abreu, and M. D'Amorim, ''Prevalence of single-fault fixes and its impact on fault localization,'' in *Proc. IEEE Int. Conf. Softw. Test., Verification Validation (ICST)*, Mar. 2017, pp. 12–22.

[25] M. Renieres and S. P. Reiss, ''Fault localization with nearest neighbor queries,'' in *Proc. 18th IEEE Int. Conf. Autom. Softw. Eng.*, Oct. 2003, pp. 30–39.

[26] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, ''Lightweight fault-localization using multiple coverage types,'' in *Proc. IEEE 31st Int. Conf. Softw. Eng. (ICSE)*, May 2009, pp. 56–66.

[27] Y. Wang, Z. Huang, Y. Li, and B. Fang, ''Lightweight fault localization combined with fault context to improve fault absolute rank,'' *Sci. China Inf. Sci.*, vol. 60, no. 9, p. 092113, 2017.

[28] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, ''Effective software fault localization using an RBF neural network,'' *IEEE Trans. Rel.*, vol. 61, no. 1, pp. 149–169, Mar. 2012.

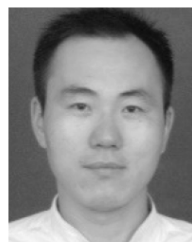
[29] W. E. Wong, V. Debroy, R. Gao, and Y. Li, ''The DStar method for effective software fault localization,'' *IEEE Trans. Rel.*, vol. 63, no. 1, pp. 290–308, Mar. 2014.

[30] X. Xie, T. Y. Chen, and B. Xu, ''Isolating suspiciousness from spectrum-based fault localization techniques,'' in *Proc. 10th Int. Conf. Quality Softw. (QSIC)*, Jul. 2010, pp. 385–392.

[31] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, ''Capturing propagation of infected program states,'' in *Proc. 7th Joint Meeting Eur. Softw. Eng., Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 43–52.

YONG WANG received the B.S. degree in computer science from Anhui Polytechnic University. He is currently pursuing the Ph.D. degree with the Computer Science Department, Nanjing University of Aeronautics and Astronautics, Nanjing. His current research interests include software testing, fault localization, and program debugging.

ZHIQIU HUANG received the Ph.D. degree in computer science from the Nanjing University of Aeronautics and Astronautics. He is currently a Professor, and the Director of software safety in computer science with the Nanjing University of Aeronautics and Astronautics. His current research interests include software safety and program debugging.



BINGWU FANG received the B.S. degree in computer science from the University of Science and Technology of China. He is currently pursuing the Ph.D. degree with the Computer Science Department, Nanjing University of Aeronautics and Astronautics, Nanjing. His current research interests include software testing, fault localization, and program debugging.

YONG LI received the B.S. degree in computer science from Northwest Minzu University. He is currently pursuing the Ph.D. degree with the Computer Science Department, Nanjing University of Aeronautics and Astronautics, Nanjing. His current research interests include empirical software engineering and machine learning.

• • •