An Approach to Generate Effective Fault Localization Methods for Programs

Babak Bagheri, Mohammad Rezaalipour, Mojtaba Vahidi-Asl

An Approach to Generate Effective Fault Localization Methods for Programs

Babak Bagheri, Mohammad Rezaalipour, and Mojtaba Vahidi-Asl(✉)

Faculty of Computer Science and Engineering, Shahid Beheshti University G. C., Tehran, Iran
mo_vahidi@sbu.ac.ir

Abstract. Software Debugging is a tedious and costly task in software development life-cycle. Thus, various automated fault localization approaches have been proposed to address this problem, among which, spectrum-based fault localization has attracted a lot of attention. Using various formulas, known as ranking metrics, spectrum-based fault localization techniques assign scores to the entities of programs (e.g., statements) based on their suspiciousness of being the root cause of failures. Despite the obvious advantages of spectrum-based fault localization techniques, such as being lightweight, they cannot effectively locate faults in every program owing to the fact that they do not consider the characteristics of the programs. We believe that program characteristics can be helpful at finding the right ranking metrics for programs, and they can assist at combining several existing ones to produce a customized ranking metric specific to a given program.
In this paper, we have proposed an approach which combines 40 different ranking metrics to generate a new ranking metric specific to a given program. Employing mutation testing operators, the proposed approach retrieves information from the program and then, using different preferential voting systems, it combines various ranking metrics based on the collected information. We have evaluated our approach on 154 faulty versions from eight different programs of Space and Siemens test suite and compare it with nine state-of-the-art ranking metrics. The experimental results indicate that the ranking metrics generated by our approach is superior with respect to evaluation metrics such as the *Exam* score and *TOP-N*.

Keywords: Software fault localization · Spectrum-based fault localization · Mutation testing · Ranking metric · Preferential Voting System

1 Introduction

Manual debugging is a difficult task that consumes a lot of resources in software development process [21]. It is reported that up to 80% of the total software development budget might be consumed by debugging tasks [20]. To address this problem, a wide variety of *Automated Fault Localization* (AFL) techniques

have been established in the literature to assist developers at locating the root causes of failures [23]. There are several approaches to automated fault localization such as slicing-based [27,12,22], machine-learning-based [26,30,14], and spectrum-based fault localization [19,10,6,1,28]. The *Spectrum-based Fault Localization* (SFL) approach has been shown to be competitive compared to the rest [16]. Also, SFL is a lightweight approach, and it can be applied to large-scale programs [29].

SFL techniques execute a given program with an existing set of passing and failing test cases. Then, leveraging *program spectra* [23] (i.e. program execution traces of test cases), and employing a *ranking metric* [26], the *suspiciousness scores* of *program entities* are computed. Program entities are source code elements with any granularity such as statements, methods, and basic blocks. Suspiciousness scores indicate the likelihood of each program entity to be faulty, and ranking metrics assign higher suspiciousness scores to entities covered by more failing tests and fewer passing ones. After the computation of suspiciousness scores, program entities are sorted according to their suspiciousness scores and handed to developers or *automated program repair* techniques [13]. Finally, the source code is examined from the most suspicious entity to the least suspicious one with the purpose of diagnosing the root causes of failures.

Several SFL techniques such as *Ochiai* [1] exist in the literature, each of which performs effectively on specific programs while not ranking entities of other programs, appropriately [28]. In other words, for most programs, current techniques assign higher suspiciousness scores to program entities that are not related to the fault at hand [12]. Our intuition is that this issue can be addressed if program characteristics are considered while suspiciousness scores are computed, which is also mentioned by Wong et al. [23]. The semantics and structures of programs are two examples of program characteristics. We believe that program characteristics can lead us toward finding right SFL techniques (among the existing ones) for any given program. Also, we hypothesize it can assist us at combining various existing ranking metrics (i.e., SFL techniques) to produce more effective ranking metrics, explicitly customized for a given program.

In this paper, we present an approach that combines various ranking metrics to generate an effective one for a given program. In this approach, first, using *mutation testing* [9], several mutants are produced for the given program which are then executed by an existing test suite. Then, runtime data such as program spectra generated for the mutants are collected which are employed as a representation of program characteristics. Afterward, these runtime data are utilized to compute the effectiveness of 40 state-of-the-art ranking metrics. In the end, considering the effectiveness calculated for these ranking metrics and employing *preferential voting systems* [4,11,2,18], the 40 ranking metrics are combined to generate a new ranking metric. We evaluate our approach using 154 faulty versions of the Siemens suite and the Space program and compare it with nine state-of-the-art SFL techniques. According to the experimental results, the ranking metrics produced by our approach always perform more effective com-

pared to the nine comparative ranking metrics, regarding well-known evaluation metrics such as the *Exam* score and *TOP-N*.

The remainder of this paper is structured as follows: Section 2 reviews preliminary materials and related work; Section 3 presents the proposed approach of this paper; Section 4 provides the experimental results and discussions; Section 5 concludes this work.

2 Background and Related Work

In the following, Section 2.1 provides a brief description of the preliminary materials related to our work, and Section 2.2 reviews some of the state-of-the-art automated fault localization techniques.

2.1 Preliminaries

Spectrum-Based Fault Localization. The goal of Spectrum-based Fault Localization (SFL) techniques is to locate faulty program entities such as statements, methods, and basic blocks. SFL techniques take as input a faulty program and two sets of test cases. One of these sets contains failing test cases while the other set has passing ones. Afterward, it collects program execution traces of the test cases, referred to as program spectra [23], by instrumenting and executing the given program, using the failing and passing test cases. Each program spectrum reports information regarding program entities that are executed by a test case. Various tools can record program spectra. For instance, in our experiments, we use *Gcov* [8] to instrument programs and retrieve runtime data. Based on program spectra, several statistics are computed for each program entity e_j such as $N_{CF}(e_j)$, $N_{CS}(e_j)$, $N_{UF}(e_j)$, and $N_{US}(e_j)$, which are the number of failing and passing (successful) test cases covering e_j, and the number of failing and passing test cases not covering e_j, respectively. Using these statistics, and employing a ranking metric [26] such as Ochiai [1], which is shown in Eq. (1), SFL techniques compute the suspiciousness score of every program entity. After computing the suspiciousness scores, the program entities are sorted and handed to developers or automated program repair techniques [13] to assist them in their debugging task.

$$Score_{Ochiai}(e_j) = \frac{N_{CF}(e_j)}{\sqrt{(N_{CF}(e_j) + N_{UF}(e_j)) \times (N_{CF}(e_j) + N_{CS}(e_j))}} \quad (1)$$

Mutation Testing. As a testing technique, mutation testing [9] is used to measure the effectiveness of test suites regarding their ability to detect faults in programs. This technique produces several mutants p_i $(1 < i < m)$ for a program p by seeding it with m faults. Faults are seeded by employing *mutation operators*, which perform syntactical modifications to programs, such as replacing a relational operator by another one. Then, the mutants are executed against the whole test suite. If the result or behavior of a mutant p_i is different compared to p, p_i is said to be killed. The higher the number of killed mutants, the more

effective the test suite is. Besides being the most successful metric to measure test suite effectiveness [3], mutation testing can be used for other purposes, as well. For example, state-of-the-art automated program repair techniques such as *ELEXIR* [17] apply various mutation operators for patch generation. In this paper, we use mutation testing to measure fault localizing capability of SFL techniques for a given program. We generate several mutants for the program at hand and then, compute the effectiveness of SFL techniques at finding the faults in these mutants.

Preferential Voting System. *Ranked voting* refers to special electoral systems in which voters can vote for more than one candidate and sort them in their ballots in order of their preferences. This type of ballot, referred to as *ranked ballot*, contains more information compared to those that only mention one candidate. Therefore, they must be processed and aggregated using certain methods specific to them called preferential voting systems. There are various preferential voting systems in the literature each of which is subject to criteria such as *monotonicity* which states that when a candidate is the winner of the election, changing a ballot in favor of this candidate must still keep it as the winner of the election. Reviewing these criteria and the advantages of different preferential voting systems are beyond the scope of this paper, and we encourage interested readers to refer to [5] for further details. For this research, we choose four preferential voting systems *Instant Run-Off Voting* [4], *Kemeny-Young* [11], *Condorcet* [2], and *Schulze* [18] because of their popularity among researchers. We use these systems to aggregate ranking ballots produced by different mutants which act as voters that prioritize various SFL techniques (ranking metrics) in their ballots.

2.2 Automated Fault Localization Techniques

There are hundreds of studies about Automated Fault Localization (AFL) techniques [23]. Program slicing-based AFL techniques obtain a *slice* for a given program by collecting its executable statements that might have an impact on the value of a specified variable. Xuan and Monperrus [27] proposed a method called *test case purification* which utilizes program slicing to reduce failing test cases with several assertions into several test cases with only one assertion. They also indicated that employing test case purification improves the fault detection capability of spectrum-based fault localization techniques. Mao et al. [12] proposed a novel approach which first employs program slicing to identify program entities that affect the given program output, and then, it uses a spectrum based fault localization technique to rank the remaining entities with respect to their suspiciousness. Wang et al. [22] presented a debugging framework called *DrDebug* that enables users to debug multi-threaded programs while focusing on a specific slice.

Machine learning, a field of artificial intelligence, has been used in various studies on different software engineering tasks such as automated program repair [17] It has also been used in automated fault localization. Xuan and Monperrus [26] employed machine learning to present a fault localization technique

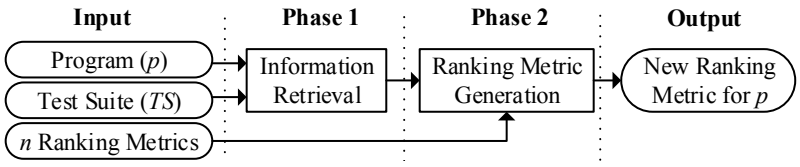

Fig. 1: Overall structure of the proposed approach

that estimates the suspiciousness of program entities by automatically combining 25 ranking metrics. Zhang and Zhang [30] employed a Markov logic network to compute the suspiciousness of program statements. Nath and Domingos [14] presented a probabilistic-based fault localization technique that finds faults according to the bug patterns it learns. This technique has the capability of employing the output of spectrum-based fault localization techniques as features, and can be trained on a set of faulty programs.

Spectrum-based fault localization is probably the most studied approach in the field, which is thoroughly reviewed in [19]. The first ranking metric, Tarantula, was proposed by Jones et al. [10] which is based on the idea that program entities covered by more failing and fewer passing test cases are the most suspicious ones of being the root causes of failures. Dallmeier et al. [6] proposed Ample as a plug-in for the Java IDE Eclipse to locate faults in object-oriented programs. Abreu et al. [1] studied three widely used ranking metrics Tarantula, Ample, and Ochiai and reported that Ochiai outperforms the other two techniques. Yoo et al. [28] studied different ranking metrics and realized that some of them are equivalent and do not dominate each other. They also concluded that there is not a ranking metric that outperforms all the other ranking metrics for every program.

The proposed approach of this paper is not based on program slicing and does not employ machine learning. Our approach combines several existing SFL techniques using preferential voting systems and mutation testing. In this regard, it is different from the studies mentioned above.

3 Proposed Approach

This section presents the proposed approach of this paper, by which an effective ranking metric is produced for a given program. As illustrated in Fig. 1, the proposed approach receives three different inputs: 1) a program p, for which a new ranking metric is produced; 2) a test suite TS; 3) n existing ranking metrics. Following two phases, the proposed approach generates a new ranking metric for p by combining the n given ranking metrics.

At the first phase, various mutation operators are applied to p to generate m mutants for it. Then, the mutants are executed by every test case in TS, and the execution results are collected and passed to the second phase (see more details in Section 3.1). At the second phase, for each mutant, the effectiveness of every n ranking metric is computed, employing the output of the first phase. Then,

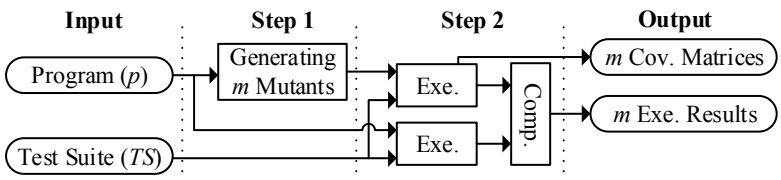

Fig. 2: Details of phase 1

these ranking metrics are combined based on their effectiveness so that a new ranking metric is generated that is more effective for p, compared to each of the n given ranking metrics, individually (see more details in Section 3.2).

3.1 Phase 1: Information Retrieval

The proposed approach generates ranking metrics specific to a given program. To this end, the characteristics of the given program must be retrieved and taken into consideration. The purpose of this phase is to collect this information, employing mutation testing. As illustrated in Fig. 2, this phase comprises two steps.

Step 1: Mutant Generation. At this step, m mutants are generated for the given program p, subject to three criteria: 1) the test suite TS must be capable of killing them all; 2) the mutants must be free of any infinite loops; 3) executing the mutants on TS must not result in any crashes or runtime errors. Those mutants that do not satisfy the criteria, mentioned above, are thrown away, and new mutants are generated to replace them. The following mutation operators are randomly used to seed a fault in a randomly selected statement:

- modifying a character or numerical literal
- changing a relational operator (e.g., >)
- changing a logical operator (e.g., &&)
- replacing a function call by another one with the same signature
- replacing a variable by another variable of the same type
- inserting a statement
- replacing a predicate with *TRUE* or *FALSE*.

Step 2: Execution. At this step, each mutant, produced at the previous step, is executed by every test case in TS. As a result, for each mutant, a matrix is produced known as program spectra for that mutant, and we refer to it as the *coverage matrix*. The output produced after executing each mutant using TS is also collected. By comparing a mutant's output with the output produced for p, the execution results for that mutant is obtained. For instance, Fig. 3 shows an example of a coverage matrix collected for a mutant, along with its execution results. Column one shows the five test cases within the test suite. Column two through eight illustrate the coverage matrix, where 0s and 1s indicate that program entity e_i is covered and not covered, respectively, while executed by test case t_i. Column nine contains the execution results for the mutant, where 0s and 1s indicate that test case t_i is failed and passed, respectively.

	e_1	e_2	e_3	e_4	e_5	e_6	e_7	r
t_1	1	1	1	1	0	1	0	0
t_2	0	0	1	0	1	1	1	0
t_3	1	0	1	1	1	1	1	0
t_4	1	0	0	1	1	1	0	1
t_5	1	1	1	0	1	0	1	1

Fig. 3: Example of a coverage matrix and execution results produced for a mutant.

3.2 Phase 2: Ranking Metric Generation

According to Fig. 4, phase 2 comprises three different steps. Following these steps, the n given ranking metrics are combined to produce a new ranking metric for p.

Step 1: Generating Ranked Ballots. At this step, for each of the m mutants, the effectiveness of the n ranking metrics are computed, using the coverage matrices and execution results produced at the previous phase. By doing so, m ranked ballots are produced, each of which contains the n ranking metrics listed according to their effectiveness at locating the fault within the corresponding mutant. Table 1 illustrates an example of 45 ranked ballots produced for a program, while $m = 45$, and $n = 5$. Column 2 and 4 show the ranked ballots, and column 1 and 3 indicate the number of instances of each ballot. For example, according to this table, for five different mutants, the sequence "$T_1 > T_2 > T_3 > T_4 > T_5$" has been produced as the ranked ballot. This ballot states that T_1 and T_5 are the most and the least effective ranking metrics at locating the faults within these five mutants.

Step 2: Selecting Ranking Metrics. At this step, the ranked ballots produced at the previous step are aggregated into an ordered list of ranking metrics, using one of the two preferential voting systems *Instant Run-Off Voting* [4] and *Kemeny-Young* [11]. For instance, applying Instant Run-Off Voting to Table 1 produces "$T_2 > T_3 > T_1 > T_4 > T_5$", and using Kemeny-Young results in "$T_4 > T_3 > T_1 > T_5 > T_2$". Then, as the output of this step, k best ranking metrics are selected among the resulting list, which is referred to as B. For example, for $k = 4$, using Instant Run-Off Voting and Kemeny-Young results in B=[T_1,T_2,T_3,T_4] and B=[T_1,T_3,T_4,T_5], respectively.

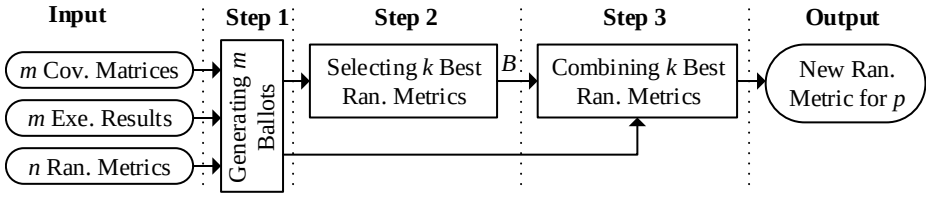

Fig. 4: Details of phase 2

Table 1: Example of ranked ballots produced at step 1 of phase 2

No. of Mutants	Ranked ballot	No. of Mutants	Ranked ballot
5	$T_1 > T_2 > T_3 > T_4 > T_5$	7	$T_2 > T_1 > T_4 > T_3 > T_5$
5	$T_1 > T_5 > T_4 > T_2 > T_3$	2	$T_2 > T_3 > T_1 > T_5 > T_4$
8	$T_3 > T_4 > T_5 > T_1 > T_2$	7	$T_5 > T_2 > T_4 > T_3 > T_1$
3	$T_2 > T_1 > T_3 > T_4 > T_5$	8	$T_4 > T_3 > T_1 > T_5 > T_2$

Step 3: Combining Ranking Metrics. As illustrated in Fig. 4, this step receives B, which contains the k best ranking metrics selected at the second step. It also gets the ranked ballots produced at the first step. Then, using Eq. (2), a new ranking metric is generated, which is the output of the proposed approach.

$$NewScore(e_j) = \sum_{i=1}^{k} w_{B_i} \times NormScore_{B_i}(e_j) \tag{2}$$

In Eq. (2), e_j represents program entities in p, for which suspiciousness scores are computed; the term w_{B_i} is the weight computed for ranking metric B_i according to its effectiveness at locating faults in p; the term $NormScore_{B_i}(e_j)$ is the normalized suspiciousness score computed by ranking metric B_i for e_j, employing the *feature scaling* method presented in Eq. (3).

$$NormScore_T(e_j) = \frac{Score_T(e_j) - min_T}{max_T - min_T} \tag{3}$$

Eq. (3) standardizes the range of suspiciousness scores that a given ranking metric (T) computes by scaling them in the range $[0, 1]$. The term $Score_T(e_j)$ is the suspiciousness score computed by T for program entity e_j; the terms min_T and max_T are respectively the minimum and maximum suspiciousness scores computed by T for all of the program entities in p.

The terms w_{B_i} $(1 < i < k)$ in Eq. 2 are determined by employing one of the two preferential voting systems *Condorcet* [2] and *Schulze* [18], and using the ranked ballots produced at the first step. In case of using Condorcet, first, Condorcet's pairwise matrix of the given ranked ballots is produced which indicates the number of times each ranking metric has been more effective compared to the rest of them. Fig. 5a shows an example of a pairwise matrix computed for the ballots in Table 1. Afterward, the terms w_{B_i} $(1 < i < k)$ are calculated using Eq. (4), where M is Condorcet's pairwise matrix. For instance, considering Fig. 5a as the pairwise matrix, w_{B_1}, w_{B_2}, w_{B_3}, and w_{B_4} are $\frac{68}{270} = 0.251$, $\frac{72}{270} = 0.266$, $\frac{59}{270} = 0.218$, and $\frac{71}{270} = 0.262$, respectively.

$$w_{B_i} = \frac{1}{\sum_{i=1}^{k} \sum_{j=1, i \neq j}^{k} M[i,j]} \sum_{j=1, j \neq i}^{k} M[i,j] \tag{4}$$

In case of using Schulze, first, Schulze's strength matrix is computed for the given ranked ballots. This matrix illustrates the strengths of the strongest

		Against				
		$\mathbf{B_1}$	$\mathbf{B_2}$	$\mathbf{B_3}$	$\mathbf{B_4}$	**Total**
For	$\mathbf{B_1}$	-	26	20	22	68
	$\mathbf{B_2}$	19	-	29	24	72
	$\mathbf{B_3}$	25	16	-	18	59
	$\mathbf{B_4}$	23	21	27	-	71
						270

(a)

		Against				
		$\mathbf{B_1}$	$\mathbf{B_2}$	$\mathbf{B_3}$	$\mathbf{B_4}$	**Total**
For	$\mathbf{B_1}$	-	26	26	24	76
	$\mathbf{B_2}$	25	-	29	24	78
	$\mathbf{B_3}$	25	25	-	24	74
	$\mathbf{B_4}$	25	25	27	-	77
						305

(b)

Fig. 5: Example of a pairwise and strength matrix produced for the lists in Table1. (a) Pairwise matrix; (b) Strength matrix.

paths for each pair of ranking metrics. In other words, it indicates how effective a ranking metric has performed compared to the other ranking metrics (for further details on strongest paths refer to [18]). Then, the weights are computed employing Eq. (4), where M is Schulze's strength matrix. Fig. 5b indicates an example of a strength matrix calculated for the lists in Table 1. Using this matrix, w_{B_1}, w_{B_2}, w_{B_3}, and w_{B_4} are $\frac{76}{305} = 0.249$, $\frac{78}{305} = 0.255$, $\frac{74}{305} = 0.242$, and $\frac{77}{305} = 0.252$, respectively.

4 Experiments

In this section, we present the evaluation of the proposed approach. Section 4.1 reviews the experiment setup; Section 4.2 provides the results of the experiments; Section 4.3 presents the discussion; Section 4.4 explains the threats to the validity of the experimental results.

4.1 Experiment Setup

Subject Programs. The proposed approach is evaluated on eight popular programs, the Siemens suite along with the Space program, provided by *Software-artifact Infrastructure Repository* (SIR) [7], which has been employed by various fault localization studies. Table 2 illustrates the details of these programs. The first row shows each program's size. Row two indicates the number of faulty versions we have used in our experiments, each of which contains a single bug. Row three shows the size of each program's test suite, and row four illustrates the number of mutants generated for the programs, which is the parameter m of the proposed approach. During the experiments, we made sure that the generated mutants were different from their corresponding faulty versions by analyzing them, manually.

Evaluation Metrics. To evaluate the effectiveness of the proposed approach, we used three metrics of evaluation, which are defined as follows:

1. Exam: The *Exam* score [23] indicates the percentage of code that needs to be inspected to locate the fault within a program. This metric is used to

Table 2: Subject programs.

	Print tokens	Print tokens 2	Replace	Schedule	Schedule 2	Tcas	Tot info	Space	**Total**
LOC	478	399	512	292	301	141	440	6218	**8781**
No. of Faulty Versions	5	10	31	9	9	36	19	35	**154**
No. of Test Cases	4130	4115	5542	2650	2710	1608	1051	13858	
No. of Mutants (m)	143	127	203	118	126	96	183	483	

compare AFL techniques on a single program while in our experiments, we had 154 faulty versions of eight different programs (see Table 2). Therefore, for any ranking metric T, we computed T's *Exam* score on every faulty version, and then, reported the mean of these 154 resulting scores as the *Exam* score of T. A lower value of this metric indicates higher effectiveness.

2. Proportion of Located Faults: This evaluation metric indicates the percentage of faults located while a specific percentage of program entities are inspected. To compute this metric for a ranking metric T, the top 10% of the program entities in each faulty version were inspected, and the percentage of located faults was reported. A higher value of this metric indicates higher effectiveness.

3. TOP-N: This metric is similar to the previous one with the only difference that in this metric, instead of a specific percentage of program entities, a certain number of them are inspected. Considering the fact that regardless of the size of programs, developers usually inspect a few of the top-ranked program entities presented by AFL techniques [15], this metric is important in practice. In our experiments, to compute this metric for a ranking metric T, top ten program entities in each faulty version were examined, and the number of located faults were reported as T's *TOP-10* score. Note that a higher value of this metric indicates higher effectiveness.

Configuration and Implementation. We utilized the 40 state-of-the-art ranking metrics presented in Table 3 as the third input to the proposed ap-

Table 3: Ranking metrics used in the experiments.

#	Name	#	Name	#	Name	#	Name
1	Braun-Banquet [23]	11	Ample [6]	21	Hamming [23]	31	Sorensen-Dice [23]
2	Baroni-Urbani & Buser [23]	12	Phi (Geometric Mean) [23]	22	Hamann [23]	32	Tarantula [10]
3	Mountford [23]	13	Arithmetic Mean [23]	23	Sokal [23]	33	Naish2 [24]
4	Fossum [23]	14	Cohen [23]	24	Scott [23]	34	Ochiai [1]
5	Pearson [23]	15	Fleiss [23]	25	Rogot1 [23]	35	Wong [24]
6	Gower [23]	16	Zoltar [23]	26	Kulczynski [23]	36	GP13 [25]
7	Michael [23]	17	Harmonic Mean [23]	27	Anderberg [23]	37	GP02 [25]
8	Pierce [23]	18	Rogot2 [23]	28	Dice [23]	38	GP03 [25]
9	Dennis [23]	19	Simple Matching [23]	29	Goodman [23]	39	GP19 [25]
10	Tarwid [23]	20	Rogers & Tanimoto [23]	30	Jaccard [23]	40	Russel & Rao [24]

proach, and thus, in our experiments, n was always 40. As stated in Section 3, for the task of selecting k best ranking metrics, the proposed approach can use one of the two methods *Instant Run-Off Voting* and *Kemeny-Young*, and to perform the task of combining these ranking metrics, it may employ *Condorcet* or *Schulze*. As a result, four different instances of the proposed approach was implemented, each of which utilizes one of the two possible preferential voting systems for these two tasks. These four instances are: "Instant Run-Off Voting + Condorcet", "Instant Run-Off Voting + Schulze," "Kemeny-Young + Condorcet," and "Kemeny-Young + Schulze," which we refer to as IRV-C, IRV-S, KY-C, and KY-S, respectively.

All of the four instances of the proposed approach were implemented in C++, and the experiments were conducted on a virtual machine with Intel Core i5 CPU at 1.60 GHz, 2GBs of RAM, and the 64-bit version of Ubuntu 16.04. To instrument code and retrieve runtime data, we employed *Gcov* [8], the GNU coverage testing tool that considers code lines as program entities.

4.2 Results

In this section we present the results of comparing the four instances of the proposed approach namely KY-S, IRV-S, KY-C, and IRV-C with nine state-of-the-art ranking metrics $Naish2$ [24], $Zoltar$ [23], $GP13$ [25], $Ochia$ [1], $Tarantula$ [10], $Jaccard$ [23], $GP03$ [25], $GP02$ [25], and $Wong$ [24]. Fig. 6a shows the results of the effectiveness comparison with respect to the first evaluation metric presented in Section 4.1. According to the results, the four instances of the proposed approach perform better than the rest of the ranking metrics. Also, KY-S shows better effectiveness compared to the other three instances of the proposed approach. Furthermore, the results indicate that fault localization effectiveness can be increased by up to 62% using KY-S.

Fig. 6b compares the effectiveness of the proposed approach with other ranking metrics with respect to the second evaluation metric presented in Section 4.1. The purpose of this experiment is to evaluate the proposed approach while only a small portion of program entities (in our case 10% of them) are examined, which is an important perspective since developers tend not to examine every program entity presented by AFL techniques. Based on the results, the proposed approach has the best effectiveness compared to other ranking metrics, and again, KY-S performs better than the other three instances of the proposed approach. To further investigate the effectiveness of the proposed approach, we also compared KY-S, and KY-C with Naish2, Ochiai, and GP13 while the portion of inspected program entities varied from 20% to 50%, and the results are illustrated in Fig. 6c. As can be seen, no matter how many program entities are inspected, KY-S is always superior.

Fig. 6d shows the results of comparing the proposed approach with other ranking metrics, regarding the third evaluation metric presented in Section 4.1, which indicates the number of located faults by each ranking metric while only ten program entities are inspected. According to the results, KY-S is superior compared to the other ranking metrics.

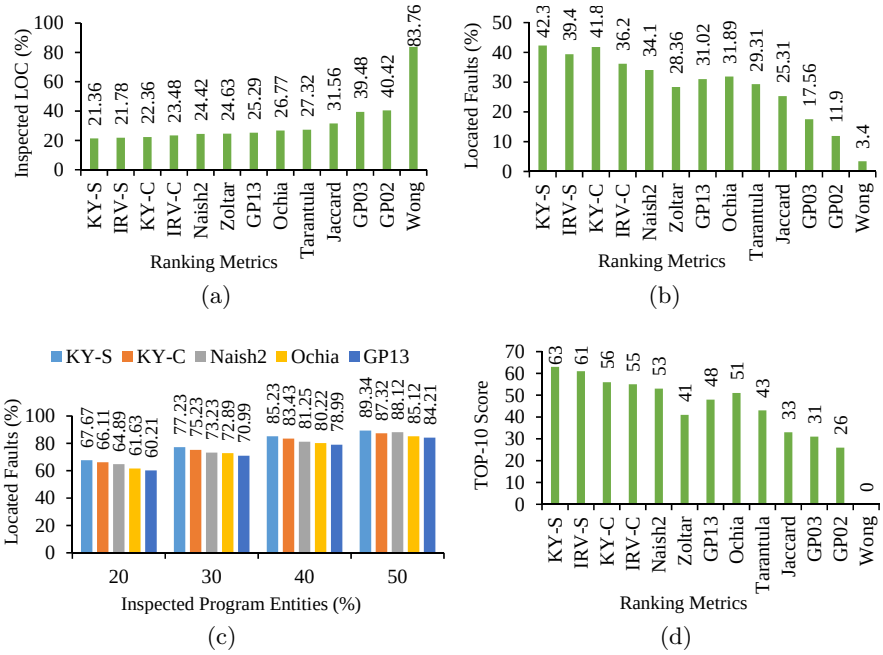

Fig. 6: Experimental results. (a) *Exam* scores; (b) proportion of located faults; (c) proportion of faults located with respect to inspected program entities; (d) *TOP-10* scores.

4.3 Discussions

According to the experimental results presented in Section 4.2, the preferential voting system used at step 3 in phase 2, which combines the best ranking metrics, has a significant impact on the effectiveness of the generated ranking metric. Considering the experimental results, employing the Schulze method results in ranking metrics that are more effective than those produced by the Condorcet method. We believe that this advantage is rooted in the ability of Schulze in considering the transitive relation between the ranking metrics in ranked ballots produced at step 1 in phase 2. In other words, compared to Condorcet, the Schulze method can more appropriately determine the effectiveness of different ranking metrics based on given ranked ballots.

Another important factor for generating an effective ranking metric is the preferential voting system employed at step 2 in phase 2, which selects k best ranking metrics among n. To investigate the impact of this factor, we removed this step by setting k to n, and then, repeated the experiments. By doing so, the *Exam* score of KY-S growed from 21.36% to 31.54% (which demonstrates a decline in its effectiveness), and also the effectiveness of KY-S with respect to the second and the third evaluation metrics presented in Section 4.1 decreased

Table 4: Sensitivity analysis of the parameter k for KY-S.

	$k = 5$	$k = 20$	$k = 40$
Exam score (%)	21.36	29.46	31.45
Proportion of faults located (%)	42.3	24.3	16.8

from 42.3% to 16.8%, and from 63 bugs to 28 bugs, respectively. The parameter k also has a significant influence on the effectiveness of the proposed approach. To investigate the impact of this parameter, we repeated the experiments by setting k as 5, 20, and 40. The results of this experiment is illustrated in Table 4, according to which KY-S has the best effectiveness for $k = 5$.

4.4 Threats to Validity

The most critical threat to the validity of our experimental results is whether they generalize to other programs. We have evaluated the proposed approach using the Siemens suite which comprises relatively small programs. However, these programs have been employed by many researchers in the field, and also, we tried to mitigate this issue by using 35 faulty versions of the Space program which are quite larger compared to the items in the Siemens suite.

In addition, the type of mutants generated at step 1 in phase 1, and the number of ranking metrics selected at step 2 in phase 2 (the parameter k) can also affect the experimental results, and thus, they are considered as other threats to the validity of our results.

5 Conclusions

In this paper, we presented an approach to generate SFL ranking metrics for programs by combining various existing ranking metrics. We implemented four instances of the proposed approach based on the preferential voting systems used for two different tasks within the approach. All four instances of the proposed approach were evaluated using the Siemens suite and the Space program, and they were compared with nine state-of-the-art ranking metrics. According to the results, using Kemeny-Young for selecting the best ranking metrics and employing Schulze to combine them leads to better ranking metrics compared to the other three instances of the proposed approach. Also, all four instances of the proposed approach generate ranking metrics that are more effective than the baselines with respect to the evaluation metrics such as the *Exam* score and *TOP-N*.

In this work, we used four different preferential voting systems while there are many other such systems that we plan to investigate their impact on our approach. Also, to reduce the threat to the validity of our results, we are going to evaluate our approach on Object-oriented, real-world, and large-sized programs, as well. Since each subject program used in our experiments had only one bug, we are going to evaluate our approach on programs with multiple bugs, as well.

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: An evaluation of similarity coefficients for software fault localization. In: Proc. 12th Pacific Rim International Symposium on Dependable Computing. pp. 39–46 (2006). https://doi.org/10.1109/PRDC.2006.18
2. Amy, D.J.: Behind the ballot box: a citizen's guide to voting systems. Greenwood Publishing Group (2000)
3. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? [software testing]. In: Proc. 27th International Conference on Software Engineering. pp. 402–411 (2005). https://doi.org/10.1109/ICSE.2005.1553583
4. Cary, D.: Estimating the margin of victory for instant-runoff voting. In: Website Proc. Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (2011)
5. Cranor, L.F.: Declared-strategy voting: an instrument for group decision-making. Ph.D. thesis, Washington University (1996)
6. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight bug localization with ample. In: Proc. 6th International Symposium on Automated Analysis-driven Debugging. pp. 99–104 (2005). https://doi.org/10.1145/1085130.1085143
7. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering **10**(4), 405–435 (2005). https://doi.org/10.1007/s10664-005-3861-2
8. Gough, B., Stallman, R.M.: An Introduction to GCC for the GNU compilers gcc and g++. Network Theory Ltd (2004)
9. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering **37**(5), 649–678 (2011). https://doi.org/10.1109/TSE.2010.62
10. Jones, J.A., Harrold, M.J., Stasko, J.: Visualization of test information to assist fault localization. In: Proc. 24th International Conference on Software Engineering. pp. 467–477 (2002). https://doi.org/10.1145/581396.581397
11. Kemeny, J.G.: Mathematics without numbers. Daedalus **88**(4), 577–591 (1959), retrieved from https://www.mitpressjournals.org/loi/daed
12. Mao, X., Lei, Y., Dai, Z., Qi, Y., Wang, C.: Slice-based statistical fault localization. Journal of Systems and Software **89**, 51 – 62 (2014). https://doi.org/10.1016/j.jss.2013.08.031
13. Monperrus, M.: Automatic software repair: A bibliography. ACM Computing Surveys **51**(1), 17:1–17:24 (2018). https://doi.org/10.1145/3105906
14. Nath, A., Domingos, P.: Learning tractable probabilistic models for fault localization. In: Proc. 13th AAAI Conference on Artificial Intelligence. pp. 1294–1301 (2016)
15. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: Proc. International Symposium on Software Testing and Analysis. pp. 199–209 (2011). https://doi.org/10.1145/2001420.2001445
16. Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B.: Evaluating and improving fault localization. In: Proc. 39th International Conference on Software Engineering. pp. 609–620 (2017). https://doi.org/10.1109/ICSE.2017.62
17. Saha, R.K., Lyu, Y., Yoshida, H., Prasad, M.R.: Elixir: Effective object-oriented program repair. In: Proc. 32nd IEEE/ACM International

Conference on Automated Software Engineering. pp. 648–659 (2017). https://doi.org/10.1109/ASE.2017.8115675
18. Schulze, M.: A new monotonic, clone-independent, reversal symmetric, and condorcet-consistent single-winner election method. Social Choice and Welfare **36**(2), 267–303 (2011). https://doi.org/10.1007/s00355-010-0475-4
19. de Souza, H.A., Chaim, M.L., Kon, F.: Spectrum-based software fault localization: A survey of techniques, advances, and challenges (2016), `http://arxiv.org/abs/1607.04347`
20. Tassey, G.: The economic impacts of inadequate infrastructure for software testing. Tech. Rep. 7007.011, National Institute of Standards and Technology (2002), `https://www.nist.gov/document/report02-3pdf`
21. Thung, F., Wang, S., Lo, D., Jiang, L.: An empirical study of bugs in machine learning systems. In: Proc. 23rd International Symposium on Software Reliability Engineering. pp. 271–280 (2012). https://doi.org/10.1109/ISSRE.2012.22
22. Wang, Y., Patil, H., Pereira, C., Lueck, G., Gupta, R., Neamtiu, I.: Drdebug: Deterministic replay based cyclic debugging with dynamic slicing. In: Proc. of Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 98:98–98:108 (2014). https://doi.org/10.1145/2544137.2544152
23. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016). https://doi.org/10.1109/TSE.2016.2521368
24. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology **22**(4), 31:1–31:40 (2013). https://doi.org/10.1145/2522920.2522924
25. Xie, X., Kuo, F.C., Chen, T.Y., Yoo, S., Harman, M.: Provably optimal and human-competitive results in sbse for spectrum based fault localisation. In: Proc. 5th International Symposium on Search Based Software Engineering - Volume 8084. pp. 224–238 (2013). https://doi.org/10.1007/978-3-642-39742-4_17
26. Xuan, J., Monperrus, M.: Learning to combine multiple ranking metrics for fault localization. In: Proc. IEEE International Conference on Software Maintenance and Evolution. pp. 191–200 (2014). https://doi.org/10.1109/ICSME.2014.41
27. Xuan, J., Monperrus, M.: Test case purification for improving fault localization. In: Proc. 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 52–63 (2014). https://doi.org/10.1145/2635868.2635906
28. Yoo, S., Xie, X., Kuo, F.C., Chen, T.Y., Harman, M.: No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. Tech. Rep. Research Note RN/14/14, University College London (2014), `http://www.cs.ucl.ac.uk/fileadmin/UCL-CS/research/Research_Notes/rn-14-14_03.pdf`
29. Zhang, M., Li, X., Zhang, L., Khurshid, S.: Boosting spectrum-based fault localization using pagerank. In: Proc. 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 261–272 (2017). https://doi.org/10.1145/3092703.3092731
30. Zhang, S., Zhang, C.: Software bug localization with markov logic. In: Companion Proc. 36th International Conference on Software Engineering. pp. 424–427 (2014). https://doi.org/10.1145/2591062.2591099