Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study

Jiajun Jiang†, Ran Wang†, Yingfei Xiong†, Xiangping Chen‡, Lu Zhang†
†Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU)
†Department of Computer Science and Technology, EECS, Peking University, Beijing, China
‡Guangdong Key Laboratory for Big Data Analysis and Simulation of Public Opinion
‡The School of Communication and Design, Sun Yat-sen University, Guangzhou, China
{jiajun.jiang, wangrancs, xiongyf, zhanglucs}@pku.edu.cn, Chenxp8@mail.sysu.edu.cn

***Abstract*—Program debugging is a time-consuming task, and researchers have proposed different kinds of automatic fault localization techniques to mitigate the burden of manual debugging. Among these techniques, two popular families are spectrum-based fault localization (SBFL) and statistical debugging (SD), both localizing faults by collecting statistical information at runtime. Though the ideas are similar, the two families have been developed independently and their combinations have not been systematically explored.**

In this paper we perform a systematical empirical study on the combination of SBFL and SD. We first build a unified model of the two techniques, and systematically explore four types of variations, different predicates, different risk evaluation formulas, different granularities of data collection, and different methods of combining suspicious scores.

Our study leads to several findings. First, most of the effectiveness of the combined approach contributed by a simple type of predicates: branch conditions. Second, the risk evaluation formulas of SBFL significantly outperform that of SD. Third, fine-grained data collection significantly outperforms coarse-grained data collection with a little extra execution overhead. Fourth, a linear combination of SBFL and SD predicates outperforms both individual approaches.

According to our empirical study, we propose a new fault localization approach, PREDFL (Predicate-based Fault Localization), with the best configuration for each dimension under the unified model. Then, we explore its complementarity to existing techniques by integrating PREDFL with a state-of-the-art fault localization framework. The experimental results show that PREDFL can further improve the effectiveness of state-of-the-art fault localization techniques. More concretely, integrating PREDFL results in an up to 20.8% improvement w.r.t the faults successfully located at Top-1, which reveals that PREDFL complements existing techniques.

***Index Terms*—Software engineering, Fault localization, Program debugging**

I. INTRODUCTION

Given the cost of manual debugging, a large number of fault localization approaches have been proposed in the past two decades. Among all these approaches, two families of approaches have received significant attentions: spectrum-based fault localization (SBFL) [1]–[3] and statistical debugging (SD) [4]–[7]. Spectrum-based fault localization collects coverage information over different elements during test execution, and calculates the suspiciousness of each element using a risk evaluation formula. On the other hand, statistical debugging seeds a set of predicates into the programs, collects whether they are covered and whether they are evaluated to true during test executions, and calculates the suspiciousness of each predicate using a risk evaluation formula.

Though being similar in the sense of using runtime coverage information to find the suspicious elements, the two families have been developed mostly independently and researchers in each community explored different aspects of the approaches. In the domain of SBFL, different risk evaluation formulas have been proposed and compared both theoretically and empirically [8], [9]. In the domain of SD, different classes of predicates have been designed and studied [5]–[7], [10]–[14]. Since the basic ideas behind the two families are similar, a question naturally raises: whether is it possible to combine the strength of the two families and how would the combination perform?

To answer this question, we build a unified model that captures the commonalities of the two approaches. In this model, SBFL is treated as a kind of predicate in SD, SBFL risk evaluation formulas are mapped to SD risk evaluation formulas, and the suspicious scores of predicates are mapped back to program elements. In this way, we can combine the two approaches as a unified one and explore different variation points. Based on this model, we perform a systematic study to empirically explore four variation points.

Predicates In SD, a large number of predicates have been proposed. We explore the performance of different predicate groups, including three groups from SD and one group from SBFL. We also explore the performance of combinations of the groups.

Risk Evaluation Formulas Particularly in SBFL, a large number of risk evaluation formulas have been proposed. In this paper we evaluate how the different formulas perform over SD predicates.

Granularity of Data Collection Existing studies have seeded predicates at different granularities, e.g., at the method level or at the statement level. In this paper we explore the effect of seeding different predicates on the localization performance and the execution time.

Methods for Combining Suspicious Scores When mapping the suspicious scores from predicates to elements, we

*Yingfei Xiong is the corresponding author.

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00054

need to combine the suspicious scores of different elements into one. Here we consider two ways of combining the scores, the maximum of different predicates and the weighted sum of different predicates.

Our empirical study is carried out over Defects4J [15], a collection of real-world faults that are widely used in recent studies [16]–[19]. The empirical study leads to several findings, as listed below.

- Among all predicates, the predicates from branch conditions contribute most to the Top-1 recall of fault localization accuracy.
- Using SBFL formulas for SD predicates significantly increases the localization accuracy, a 227.9% increase in Top-1 compared with using original SD formula and a 52.7% increase compared with a revised SD formula.
- Collecting information at the statement level leads to 69.9% increase in localization accuracy with respect to Top-1 and 40.0% increase in execution time compared to the method level. However, the overall localization time is still small, i.e, less than three minutes.
- A linear combination of the suspicious scores from SBFL and SD predicates leads to the best results.

Inspired by the above findings, we proposed a new fault localization technique, PREDFL, that combines the predicates of SBFL and SD under our unified model. The evaluation results show that PREDFL could improve the state-of-the-art when further combined with other techniques. In summary, we make the following contributions in this paper.

- A unified model for combining two popular families of fault localization techniques, i.e., spectrum-based fault localization and statistical debugging.
- A systematic analysis to explore different combinations between SBFL and SD under the framework with respect to four variation points, which provides insights for future fault localization research.
- A new fault localization technique coming from the systematic analysis, PREDFL, was evaluated via a contrast experiment on a state-of-the-art fault localization framework. The results show that PREDFL complements existing techniques and could further improve their effectiveness. Especially, it achieved an up to 20.8% improvement w.r.t Top-1.

II. BACKGROUND

A. Spectrum-Based Fault Localization

Spectrum-based fault localization (SBFL) technique is one of the most popular approaches used in recent studies [20]–[23] because of its simplicity and efficiency. More concretely, when given a faulty program and a set of test cases in which at least one test failed, a typical spectrum-based fault localization approach collects coverage information for each program element while running test cases, and then employs a predefined risk evaluation formula to compute suspicious scores for program elements, which represent how likely the elements are fautly. The granularity of elements varies on demand, and common granularities include statement and method. Different spectrum-based fault localization approaches follow the same paradigm but use different formulas to compute the suspicious scores.

To make it more rigorous, we next give the general form of spectrum-based fault localization approaches formally. A program E is a set of elements. Given a program element $e \in E$, we define the following notations:

- $\mathit{failed}(e)$ denotes the number of failed test cases that cover program element e.
- $\mathit{passed}(e)$ denotes the number of passed test cases that cover program element e.
- *totalfailed* denotes the number of all failed test cases.
- *totalpassed* denotes the number of all passed test cases.

A risk evaluation formula is a function mapping each element to a suspicious score based on the above four values. For instance, a popular formula in SBFL technique is Ochiai [24], which is defined as below.

$$Ochiai(e) = \frac{\mathit{failed}(e)}{\sqrt{\mathit{totalfailed} \cdot (\mathit{failed}(e) + \mathit{passed}(e))}} \quad (1)$$

This formula reveals the basic idea of spectrum-based fault localization approaches: the more frequently executed by failed test cases, the bigger the suspicious score of an element; the more frequently executed by passed test cases, the smaller the suspicious score, and thus demonstrates the relationship between test cases and program elements.

An SBFL approach assigns suspicious scores to elements based on a risk evaluation formula. Given a risk evaluation formula r and a program E, a simple SBFL approach directly returns the suspicious score from r for each element, as follows.

$$SBFL^{r}_{simple}(E) = \{(e, r(e)) \mid e \in E\}.$$

Recent approaches also consider collecting information at a granularity finer than the granularity of the target elements. For example, for method-level fault localization, several recent approaches [17], [18] first calculate the suspicious scores at the statement level, and then take the highest suspicious score of the inner statements as the suspicious score of the method. To model this behavior, we introduce a granularity function $g : E \rightarrow 2^E$ that maps an element to a set of sub elements, and then compute the suspicious score for each of them. If we do not calculate suspicious scores at the sub element level, g just map an element to itself. Given a risk evaluation formula r, a granularity function g, and a program E, an SBFL approach is defined as follows.

$$SBFL^{r,g}(E) = \{(e, max_{e_i \in g(e)} r(e_i)) \mid e \in E\}$$

B. Statistical Debugging

Statistical debugging (SD) was originally proposed for remote debugging by Liblit. et al. [4], [5]. Given a faulty program, SD dynamically instruments the program with a set of predefined predicates. Then the program is deployed for execution, where the values of predicates during executions

are collected. Based on the information collected from many executions, SD identifies the important predicates that are related to the root cause of the failure.

More formally, given a program, we use a set P to denote the instances of predicates seeded into the program. Given a predicate instance $p \in P$, we use the following notations to denote the information related to the predicate instance.

- $F(p)$ denotes the number of failed executions where p is evaluated to true at least once.
- $S(p)$ denotes the number of successful executions where p is evaluated to true at least once.
- $F_0(p)$ denotes the number of failed executions where p is covered.
- $S_0(p)$ denotes the number of successful executions where p is covered.

Similar to SBFL, SD uses a formula to determine the suspiciousness of a predicate, known as the importance score in the SD context. An importance formula i for SD is a function that maps an element to an importance score based on the above values. A standard importance formula [5] is defined as follows.

$$\mathit{Importance}(p) = \frac{2}{\frac{1}{\mathit{Increase}(p)} + \frac{1}{\mathit{Sensitive}(p)}} \tag{2}$$

where

$$\mathit{Increase}(p) = \frac{F(p)}{S(p)+F(p)} - \frac{F_o(p)}{S_o(p)+F_o(p)}$$

$$\mathit{Sensitive}(p) = \frac{\log(F(p))}{\log(\mathit{totalfailed})}$$

In Equation 2, the importance formula is the harmonic mean of two compositions, $\mathit{Increase}(p)$ and $\mathit{Sensitive}(p)$, where $\mathit{Increase}(p)$ distinguishes the value distributions of predicate p for failed executions among all executions that cover p, while $\mathit{Sensitive}(p)$ captures how often failed test cases evaluate predicate p to true. The more frequently being true of predicate p in failed executions, the bigger the importance score of p. The less frequently being true of p in failed executions, the smaller the importance score of p. Hence, the importance formula reflects the relationship between test cases and the value of predicates. In the rest of the paper we shall refer the above importance formula as the SD formula. Particularly, when $\mathit{totalfailed} = 1$, the score of predicate is 0 to avoid "divide by zero" error according to the definition in the paper [5].

A typical SD approach uses three groups of predefined predicates: **branches**, **returns** and **scalar-pairs** [5].

- The **branches** group includes all conditional expressions and their negations in the program, such as the expression in an `if` statement.
- The **returns** group compares the method return value (if exists) with constant 0 with respect to a set of comparators, i.e., $>$, $<$, $\geq$, $\leq$, $==$ and $\neq$.
- The **scalar-pairs** denotes comparing the left-value of an `Assignment` with all in-scope variables or constants with the same type.

The three groups of predicates are inserted into different locations of the program. As described by the predicate definitions, the **branches** predicates are instrumented into the locations of conditional expressions, the **returns** predicates are instrumented into the locations of return statements while the **scalar-pairs** predicates are instrumented into the locations of assignments or variable declarations.

To model the seeding of predicates, we define the concept of seeding function. A seeding function s maps a program element to a set of predicate instances. By using different seeding functions, we get different sets of predicate instances.

Given an importance formula i, a seeding function s, an SD approach is a function mapping a program (a set of elements) to a set of predicate instances with their importance scores.

$$SD^{i,s}(E) = \{(p, i(p)) \mid p \in s(e), e \in E\}$$

III. The Unified Model

As we can see from the previous section, there are many commonalities between the two families of approaches, e.g., both approaches use a formula to score elements. Therefore, it is possible to build a model that combines the two approaches.

The basic idea of our model is to treat SBFL as a kind of predicate in SD. Since SBFL concerns about the coverage of an element, its behavior can be captured by a predicate "`True`", which evaluates to true whenever the element is covered. Formally, given an instance of predicate `True` at element e, $\texttt{True}^e$, we define its runtime information as follows.

$$F(\texttt{True}^e) = F_0(\texttt{True}^e) = \mathit{failed}(e)$$
$$S(\texttt{True}^e) = S_0(\texttt{True}^e) = \mathit{passed}(e)$$

Also, we would like to use the SBFL formulas for SD predicates. Since SBFL relies on function $\mathit{failed}()$ and $\mathit{passed}()$, we need to make these functions work for predicate instances. Given a predicate instance p, we define the following.

$$\mathit{failed}(p) = F(p)$$
$$\mathit{passed}(p) = S(p)$$

Please note here we use $F(p)$ and $S(p)$ instead of $F_0(p)$ and $S_0(p)$, because choosing the latter two would lead to the same suspicious score for all predicates inserted at the same elements. In other words, we view each predicate as a sub element that captures a subset of states at a specific program point. Based on this unification, we shall not distinguish risk evaluation formulas and importance formulas and address them uniformly as risk evaluation formulas.

Furthermore, SBFL and SD have different results: while SBFL gives a list of suspicious program elements, SD gives a list of important predicates. To unify the results, we return a list of suspicious elements in the unified model. We assume the existence of a high-order function that aggregates the importance scores of each predicate instance to produce the suspicious score of the program element, called combining method. A combining method c is a function that takes a seeding function s, a risk evaluation formula r and an element e, and produces the suspicious score of e. A basic combining

method is to take the maximum importance score of all predicates.

$$c(s, e, r) = max_{p \in s(e)} r(p)$$

Putting everything together, we can define the unified approach of SBFL and SD. Given a seeding function s, a risk evaluation formula r, a granularity function g, and a combining method c, a unified approach is defined as follows.

$$UNI^{s,r,g,c}(E) = \{(e, max_{e_i \in g(e)} c(s, e_i, r)) \mid e \in E\} \quad (3)$$

As we can see from equation 3, the unified approach is parameterized on four components, s, r, g, and c. In the following sections we shall systematically explore the variations of the four components.

IV. Empirical Study Setup

In this section, we systematically explore the variations of the four components in the unified model. We first explain the research questions, then introduce the experiment setup including the subjects used in the experiment and the variations for each component. Finally, we present the evaluation metrics and experiment implementation.

A. Research Questions

1) **Which kinds of predicates are most important?** After treating SBFL as a predicate in the unified model, we have an even richer set of predicates than SD approaches. In this question, we explore the effectiveness of different predicate groups and understand which kinds of predicates contributed the effectiveness most.
2) **How does the risk evaluation formula impact the effectiveness of fault localization?** Existing researches have both theoretically and empirically studied different risk evaluation formulas in the SBFL scenario [8], [9]. In this research question, we also selected a set of representative formulas to explore the effects of different ones under our unified model.
3) **How does the granularity of data collection impact the fault localization result?** Recently proposed method-level fault localization approaches collect program coverage data from two different granularities, which are statement level [17], [18] and method level [25]. Intuitively, the finer the granularity, the better localization accuracy and the longer execution time. However, this intuitive proposition has not been evaluated in existing studies, nor it is clear how much the increases at localization accuracy and execution time will be. Therefore, in this question, we aim to explore the impact of different granularity functions to the effectiveness and efficiency of fault localization approach via a contrast experiment.
4) **How does combining method among different predicates impact the effectiveness of fault localization?** In the unified model, we need to combine the suspicious scores of the predicates to form the scores of the program elements. In this research question we explore two basic ways of combination: the maximum suspicious score of all predicates and the linear combination of all suspicious scores (details in Section IV-C4).

B. Subject Projects

We conducted the experiments on the Defects4J [15] (v1.0.1) benchmark, which is a commonly used dataset in automatic fault localization [17], [25] and program repair studies [21], [23], [26]–[33]. It contains 357 real-world faults from five large open source projects in Java language. JFree**Chart** is a framework to create chart, Google **Closure** is a JavaScript compiler for optimization, Apache commons-**Lang** and commons-**Math** are two libraries complementing the existings in JDK, while Joda-**Time** is a standard time library. The subjects involve diversity applications with different scales (from 22k to 96k LOC) of source code, which is listed in Table I.

TABLE I: Details of the experiment benchmark.

Project	**#Bugs**	**#KLoC**	**#Tests**
JFree**Chart**	26	96	2,205
Apache commons-**Math**	106	85	3,602
Apache commons-**Lang**	65	22	2,245
Joda-**Time**	27	28	4,130
Closure compiler	133	90	7,927
Total	357	321	20,109

In the table, column "**#Bugs**" denotes the total number of faults in the benchmark, column "**#KLoC**" denotes the average number of thousands of lines of code in a faulty program, and column "**#Tests**" denotes the total number of test cases for each project.

C. Experiment Configuration

To answer the research questions above, we first define a set of variations for each component of our model, and together these variations form a four-dimensional design space of the combined approach. Since the whole space is large and not feasible to fully traverse, we first choose a default configuration, which was selected by a pilot study on 20 bugs randomly sampled. To ensure the representativeness of the selected bugs, we evenly and randomly sampled 4 bugs from each project in Defects4J benchmark shown in Table I. Then, each time we change one component over the default configuration in the evaluation. In the following we introduce the variations we considered for the four components, as well as the default variation we choose for each component.

In our study we focus on method-level fault localization. That is, the fault localization method assigns a suspicious value for each method. As argued by several existing studies [25], [34], method level is more suitable for developers than other granularities such as file level and statement level.

1) Predicates: To explore the performance of different predicates, we follow the original classification of statistical debugging that divides the predicates into three groups, i.e., **branches**, **returns** and **scalar-pairs**. Additionally, we consider

the predicates of SBFL as a separate group. Therefore, in total there are four groups of predicates.

Especially, when we study a specific group of predicates, the scores of predicates in other groups will be always 0 since they are not seeded. However, for other explorations, we use all the predicates (i.e., all four groups of predicates) by default.

2) Risk Evaluation Formulas: To compare the difference of effectiveness among risk evaluation formulas, we selected seven formulas in total, five from spectrum-based fault localization and two from statistical debugging. According to the definition in Section II-B, when *totalfailed* $= 1$, a prevalent case in Defects4J benchmark, the score of predicate will be 0. To mitigate this influence, besides the original SD formula, we additionally employed a formula derived from it with minor changes, called NewSD. Table II presents the definitions of all formulas used in our evaluation except SD, which is defined by Equation 2.

Particularly, we use the Ochiai, commonly used in recent studies on fault localization [16] and program repair [21], [23], [30], as the default risk evaluation formula in our experiments.

TABLE II: Formulas employed in the experiment.

Name	Formula
Ochiai [24]	$r(p) = \frac{failed(p)}{\sqrt{totalfailed \cdot (failed(p)+passed(p))}}$
Tarantula [2]	$r(p) = \frac{failed(p)/totalfailed}{failed(p)/totalfailed+passed(p)/totalpassed}$
Barinel [35]	$r(p) = 1-\frac{passed(p)}{passed(p)+failed(p)}$
DStar† **[36]**	$r(p) = \frac{failed(p)^*}{passed(p)+(totalfailed-failed(p))}$
Op2 [37]	$r(p) = failed(p) - \frac{passed(p)}{totalpassed+1}$
NewSD‡	$r(p) = \frac{2}{1/Increase(p)+\log(totalfailed)/\log(F(p)+1)}$

† The variable $*$ in the formula is greater than 0. Here, we set its value as 2, which is also employed by previous study [16].
‡ Function $Increase(p)$ in NewSD is the same as it is in Equation 2.

3) Granularity of Data Collection: To localize faults in methods, we explore two granularity functions that respectively collect predicate coverage data at method level and statement level. The granularity function at statement level maps each method to the set of statements inside the method, while the granularity function at method level maps each method to a set containing itself.

The default variation for the granularity function is the one at statement level.

4) Methods for Combining Suspicious Scores: In this paper, we use "max" or "linear" functions to combine all predicates. However, when using linear combination, the number of components should be fixed, so we linearly combine SBFL predicates with all the rest under our unified model. We call these two combining methods as MAXPRED (i.e., MAX score of PREDicates) and LINPRED (i.e., LINear score combination of PREDicates), respectively. More formally, when given a seeding function s and a risk evaluation function r, we define the combining methods as follows.

MAXPRED Given a program element e, we compute its suspicious score as the maximum score of all predicates related to it, i.e., $c(s, e, r) = max_{p \in s(e)} r(p)$.

LINPRED Given a program element e, we partition the predicates related to it into two standalone sets: P_1 and P_2, where $P_1 \cup P_2 = s(e)$ and P_1 contains one predicate from SBFL while the others constitute P_2. Then, the combining method is defined as $c(s, e, r) = (1-\alpha) \cdot max_{p \in P_1} r(p) + \alpha \cdot max_{p \in P_2} r(p)$, where $\alpha \in [0, 1.0]$.

Particularly, we set the default variation for the combining method in the experiments as LINPRED, and the default coefficient is $\alpha = 0.5$.

D. Evaluation Metrics

To evaluate the effectiveness of fault localization techniques, we employed two metrics that are usually employed by existing studies.

Recall of Top-k. This metric is used to measure how many faults can be located within top k program elements among all candidates. In a survey conducted by Kochhar et al. [34], 73% practitioners think that inspecting 5 program elements is acceptable and almost all practitioners agree that 10 elements are the upper bound for inspection within their acceptability level. Therefore, in this paper, we consider ranks within top k locations, where $k \in \{1, 3, 5, 10\}$. Moreover, we take the mean rank of a program element to break the tie when multiple elements have the same suspicious scores like exiting approaches [16], [38]–[40]. Specifically, we use ceiling function to normalize ranks to integers.

EXAM* Score**. This metric is used to measure the percentage of program elements that need to be inspected by developers among all candidates before reaching the first desired faulty element, reflecting the relative ranks of faulty elements among all candidates and the overall effectiveness of a fault localization approach. As a result, many previous studies [16], [41], [42] employed this metric as well. The smaller the ***EXAM score is, the better the result is.

E. Implementation

To collect coverage information for program element at runtime, we implemented a program instrument framework that can statically instrument programs at both statement and method level in the source code. This framework is built atop the Eclipse **J**ava **D**evelopment **T**ool (JDT) library[1], which is a Java source code manipulation framework that provides plentiful operations to Java code, such as source code deletion, insertion and replacement. Besides, it also provides the ability to parse types of variables or complex expressions, which is helpful for generating predicates of statistical debugging since those predefined predicates depend on the types of variables or expressions. Moreover, the instrumented program can preserve the semantics of the original progam and is free from side effects. Our implementation is publicly available at https://github.com/xgdsmileboy/StateCoverLocator.

[1] https://www.eclipse.org/jdt

V. Result and Analysis

In this section, we present our experimental results in detail with respect to the research questions introduced in Section IV-A.

A. Effectiveness of Individual Predicates

In this section, we explore the effectiveness of four different groups of predicates explained in Section IV-C1, and their results are shown in Figure 1, where x-axis denotes different groups of predicates, while y-axis denotes the percentages of faults that successfully located within the corresponding Top-k recall. Additionally, we present the corresponding ***EXAM*** score beneath the group names of predicates. For all following figures, we employ the same notations consistently for all figures and will not introduce them redundantly later.

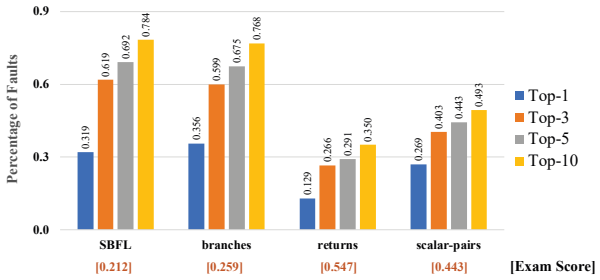

Fig. 1: Fault localization results when only employing individual group of predicates.

From the figure we can see that the predicates from **branches** achieved the best result with respect to Top-1 against any other group of predicates, but the predicates from SBFL are also effective since it achieved the best result on Top-3 and obtained the best ***EXAM*** score. Compared with the other two groups of predicates, i.e., **returns** and **scalar-pairs**, the results show that the predicates of **branches** perform significantly better with about 0.3-1.8 times improvements in terms of Top-1. Besides, the ***EXAM*** scores of both SBFL and **branches** are merit against the others. Then, we further analyzed the reason for this, and found that a majority of faults are related to existing conditional expressions, included by both SBFL and **branches**, which was also observed by previous studies [43]. Furthermore, to investigate whether different groups of predicates complement each other, we analyzed the fault localization results of several sampled combinations of different group predicates, and present their results in Figure 2.

From the figure we can find that the combination of predicates from SBFL and **branches** achieves almost the same result with the approach using all the predicates (the first column in the figure), and outperforms the other combinations, where the improvements are from 8.7% to 42.4% on Top-1. Moreover, by comparing Figure 1 and 2, the combined approaches are superior to those with standalone group of predicates in terms of either Top-1 or ***EXAM*** score, which denotes that different groups of predicates complement each other to some extent. Furthermore, the most contributors of the combined approach are predicates from SBFL and **branches** of SD. The figure also presents that the combination of these two groups of predicates performs as good as that using all predicates and better than others.

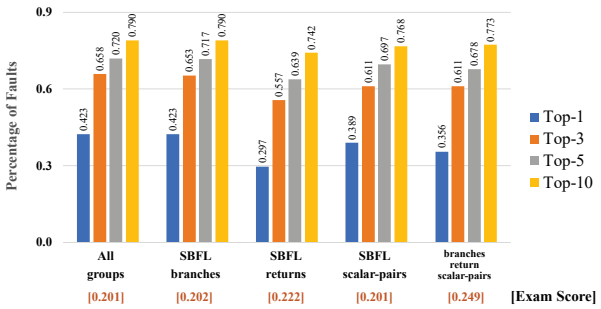

Fig. 2: Fault localization results when considering the combination among different groups of predicates.

> **Finding 1.** *Among all predicates, those from existing branch conditions and the SBFL predicates are two most important groups that contribute to the accurate fault localization result in our experiment. Moreover, existing conditions contribute most to our combined approaches with respect to Top-1.*

B. Different Risk Evaluation formulas

Figure 3 presents the fault localization results of LinPred when employing different risk evaluation formulas. This figure shows the Top-k recall and ***EXAM*** score for each formula. In the figure, "SD" denotes the risk evaluation formula of original statistical debugging defined by Equation 2. The other formulas are defined in Table II. As shown in the figure, there is no significant difference for both Top-k and ***EXAM*** score among the results except for those of "SD" and "NewSD", which is consistent with the previous studies that different risk evaluation formulas do not produce significant difference on fault localization results. However, we can notice the difference between two different kinds of formulas, i.e., formulas of SBFL and statistical debugging, where the former significantly outperform the latter in our experiment with an up to 227.9% increase at Top-1 against the original SD formula.

As explained in Section II-B, the SD formula was originally designed for remote sampling, where usually more than one failed test case exist. However, in our evaluation scenario, usually a bug is triggered by only one test case. The result comparison shows that this has a great impact on the effectiveness of SD formula since NewSD successfully located double number (26.9% vs 12.9%) of faults at Top-1 compared with the SD formula. However, it is still lower than the results of SBFL formulas, which located on average 39.7% faults at Top-1. It reveals that the formulas from SBFL are better at

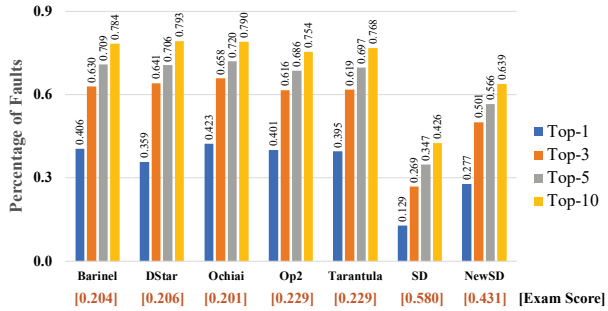

Fig. 3: Fault localization results when using different risk evaluation formulas.

distinguishing the difference of predicates among failed and passed executions in our experiment.

> **Finding 2.** *The risk evaluation formulas of SBFL significantly outperform the original SD formula in the experiment with an up to 227.9% improvements in terms of fault numbers located at Top-1.*

C. Different Granularity of Data Collection

In this section, we explore the effectiveness of fault localization appoaches under different granularity functions of data collection. Figure 4 presents the statistical results of our experiment, where the x-axis denotes the granularity of data collection, i.e., statement and method level in our evaluation.

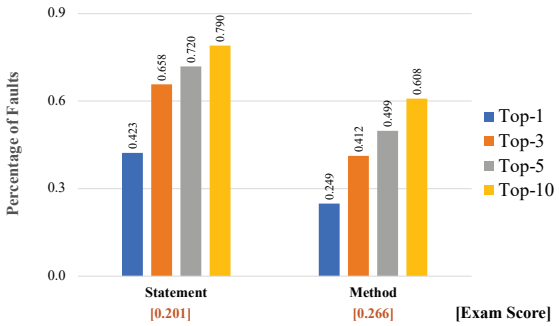

Fig. 4: Comparison of fault localization results under statement and method level data collection.

Based on the figure, the fault localization result when collecting predicate data at statement level is much better than that at method level with respect to both the recall of Top-k (k=1,3,5,10) and the ***EXAM*** score. More specifically, the former increased up to 69.9% with respect to Top-1 against the latter. This result reflects the fine-grained data collection has a more powerful capability to distinguish fault-related program elements, which is consistent with our intuition.

> **Finding 3.** *Fine-grained data collection (statement level) contributes 69.9% better fault localization result than the coarse-grained data collection (method level).*

However, scalability is also essential to fault localization approaches to make it practical. In the study conducted by Kochhar et al. [34], the satisfaction rate of practitioners increases along with the scalability of fault localization approaches. It is intuitive to us that finer granularity tend to cause larger overhead to data collection since more predicates will be considered and gathered, thus we analyzed the execution overhead under different data collection functions as well. We found that on average the predicates collected at statement level is much more than those collected at method level (about 4.1 times), while the test execution time is only 1.4 times. Moreover, the execution time of either statement or method level data collection is still less than three minutes on average (usually hours for mutation-based fault localization), which is relatively small and denotes the increase of predicates does not heavily damage the efficiency of fault localization approaches.

> **Finding 4.** *Statement-level data collection results in 4.1 times as many predicates as method-level, while uses 1.4 times as much as execution time. The overall execution time is still less than 3 minutes.*

D. Different Combining Methods

In this section, we compare the effectiveness of fault localization approaches using different combining methods under our unified model, which are MAXPRED (using "max" function) and LINPRED (using "linear" function) as introduced in Section IV-C4. We present our experimental results in Figure 5. In order to understand the improvement on effectiveness of the combined approaches against individuals, we also redundantly show the results of traditional SBFL and SD approaches in the figure.

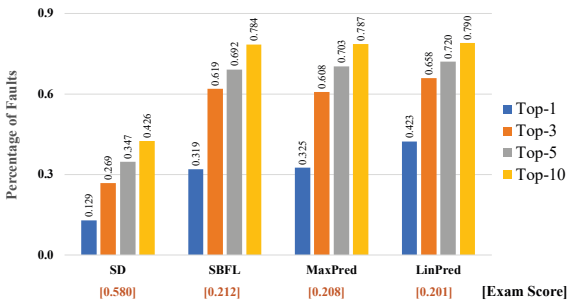

Fig. 5: Result comparison among traditional SBFL and SD approaches with the combined methods.

From the figure, the combined approaches can improve both the original SD and SBFL techniques with respect to Top-1. However, we can notice that the improvements are quite different, where the result of MAXPRED is almost the same

with SBFL while the result of LINPRED is much better than all the others with respectively 227.9% and 32.6% improvements against traditional SD and SBFL. In fact the reason is simple, on the one hand, both groups of predicates from SBFL and SD contain existing conditions, which contribute the most to Top-1 as concluded in the previous evaluation. On the other hand, both predicates from these two groups potentially contain predicates that will damage the fault localization accuracy, i.e., predicates with high suspicious scores at non-faulty locations, we call these predicates noises.

Take the fault Math-72 as an example, when only employing individual predicates (either from SBFL or SD), two candidate faulty methods (m_1 and m_2 for SBFL, m_1 and m_3 for SD) share the highest rank with the same suspicious score as 1.0, respectively, resulting in the faulty method ranked 2nd. Therefore, after taking the linear combination, LINPRED successfully ranked the faulty method m_1 at top 1 with mitigating the noises from individual sources, which cannot be overcome by MAXPRED. As a result, the improvement of LINPRED is significant but not that of MAXPRED. Moreover, the result indicates that these two sources of predicates, predicates from SBFL and SD, complement each other only if we take the proper combination of them.

As a matter of fact, in theory (see Section II-A and Section II-B) both SBFL and SD predicates have limitations. For SBFL, since all the predicates related to different program elements are constant `True`, they cannot distinguish program elements in the same execution path though the program states [44] at these program elements can be different. On the other hand, for SD, since the predicates only exist at some selective locations, e.g., the location of conditional expression, they will fail to identify those faulty elements that do not contain them. However, by combining the predicates from these two source, the predicates from SBFL can alleviate the predicate absence of SD in some program elements while the more concrete predicates from SD can assist SBFL in better distinguishing program elements in a same path.

For example, the following code snippet comes from the Math project.

```
1   public double getSumSquaredErrors() {
2 -   return sumYY-sumXY*sumXY/sumXX;
3 +   return Math.max(0d,sumYY-sumXY*sumXY/sumXX);
4   }
```

Listing 1: Faulty code snippet of Math-105 in Defects4J.

In the code snippet, the return statement in line 2 is faulty, whose repair code is shown in line 3. Individually based on the predicates from SBFL, the faulty location was ranked 8th among all the candidate locations. However, after involving the SD predicates, the ranking of the faulty location ranked 1st. The key contributor is the predicate of `sumYY-sumXY*sumXY/sumXX<0` defined by **returns** in SD, which exactly captures the root cause of the fault. Similarly, as another example, for the fault Math-53, the faulty method does not contain any predicates of SD, causing it ranked relatively low (rank 6th). However, the combined approach successfully ranked the faulty method at Top-1.

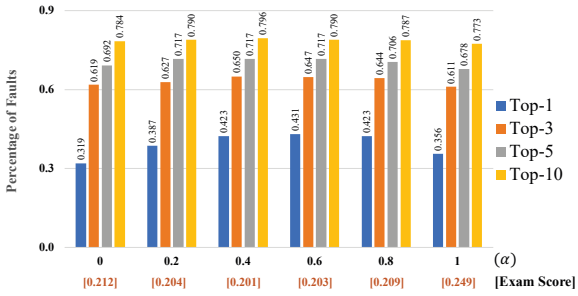

Fig. 6: The result comparison of LINPRED with different coefficients.

From the above explanation, both the complementarity and the individual limitations should be deliberated to design a relatively good combining method. Therefore, we further explored different coefficient values (α) of the linear combination and presented the results in Figure 6.

From this figure, LINPRED is always superior to the original SBFL approach (when α=0) with respect to the recall of Top-1 though the improvements varies along with the value of α. Particularly, when α falls into the interval of [0.4-0.8], LINPRED performs best. All in all, the combining approaches are relatively merit. However, we only explored several simple combinations in this paper, i.e., MAXPRED and LINPRED, more thorough investigations of potential combinations will be studied in future work.

> **Finding 5.** *The linear combination of SBFL and SD predicates (*LINPRED*) performs 23.2% better w.r.t Top-1 compared to the combination with "max" function (*MAXPRED*). Especially, when the coefficient falls into [0.4, 0.8],* LINPRED *achieves the best result.*

VI. PREDFL AND ITS PERFORMANCE

According to the previous analysis, we could find that the combined approach under the unified model is effective and better than each standalone technique. Additionally, from the experimental results, we can conclude that the combined approach with the default configurations for all variations is relatively merit in contrast with others. Therefore, we propose a new fault localization approach via instantiating the unified model with all default configurations, i.e., collecting all groups of predicates at statement level and linearly combining predicate scores computed with Ochiai formula. For convenient reference, we call this approach as PREDFL.

Although standalone fault localization approaches have gained a big success [24], [44]–[46] in the last decade, recently researchers have noticed the benefit of combining different data sources or existing fault localization techniques to improve state-of-the-art techniques. Based on the previous investigation, each standalone approach has advantages and

limitations. Therefore, by combining different kinds of techniques, it potentially could exert their strength while mitigating limitations. As a result, such combining techniques already benefit state-of-the-art fault localization approaches [17]–[19], [38]. Since different information have been utilized by these combining techniques, hence a question naturally raises, whether our approach (PREDFL) is already implicitly covered by existing techniques. In other words, is PREDFL complementary to existing techniques?

Therefore, in this section we explore the potential of PREDFL to improve state-of-the-art fault localization techniques, for which we integrated PREDFL with a state-of-the-art fault localization framework developed by Zou et al. [19], COMBINEFL for short. It has achieved better fault localization results compared with other state-of-the-art approaches, including MULTRIC [17], Savant [25], TraPT [17] and FLUCCS [18].

As a result, to check whether our approach is complementary to existing approaches, we combine our approach with this framework and check whether the combination brings any improvements. More concretely, the current implementation of COMBINEFL consists of spectrum-, mutation-, history- and IR-based fault localization techniques [24], [36], [45], [47]–[49], stacktrace analysis [50], dynamic slicing [51]–[53] and predicate switching technique [44]. Different techniques can be freely assembled. Especially, based on the time used to perform fault localization (including data collection) of different techniques, they are further classified into four levels and listed in Table III. For example, if we would like to locate the faulty elements within minutes, we can use all the techniques in Level 1 and Level 2 together. According to our empirical study results in Section V, the execution time of PREDFL is less than three minutes. Therefore, in this contrast experiment, we also selected the same execution time configuration, i.e., integrating PREDFL with those techniques in Level 2 (techniques in Level 1 were included). Additionally, we employed Level 4 as well since it contains almost all existing techniques, where we fed the results of PREDFL together with those of other techniques into the framework. Also, we performed method-level fault localization on Defects4J benchmark.

TABLE III: Different integration levels of COMBINEFL framework according to execution time.

Time Level	Family	Technique
Level 1 (Seconds)	history-based stack trace IR-based	Bugspots stack trace BugLocator
Level 2 (Minutes)	slicing spectrum-based	union, intersection and frequency Ochiai and DStar
Level 3 ($\sim$ Ten minutes)	predicate switching	predicate switching
Level 4 (Hours)	mutation-based	Metallaxis and MUSE

Table IV presents the results in detail. From the table, we can see that after integrating our approach, all the results are improved from 4.8% (75.9% vs 79.6%) to 20.8% (48.7% vs 40.3%). Furthermore, PREDFL even improves the fault localization effectiveness on each individual project w.r.t Top-1. Especially, after integrating PREDFL, the fault localization results at Level 2 are even better than those at Level 4 without PREDFL. Please note, since the current implementation of COMBINEFL is already superior to all existing techniques as evaluated in their paper, we do not redundantly compare those techniques here because the integrated approach is already more effective than the original one.

> **Finding 6.** *Our combined approach,* PREDFL *that linearly combines predicates from SBFL and SD, can further improve existing techniques, and the improvement can be up to 20.8% in terms of Top-1, which denotes our approach complements existing techniques.*

VII. IMPLICATIONS FOR FUTURE STUDY

A. Predicate Selection

From the experimental results we can see that the selection of predicates play an important role in the performance of the fault localization. On the one hand, not all predicates play an equal role in identifying the faulty locations. As our experiment results, predicate group **branches** contributes most to the accuracy of fault localization, while some predicates may even play negative roles, the performance of all predicates is sometimes lower than using the predicates from one group. On the other hand, the performance of a predicate group may vary depending on where the predicates are seeded. For example, the following code snippet in Listing 2 shows a patch for bug Chart-9 in Defects4J benchmark.

```
1   if(start == null) {
2     throw new IllegalArgumentException();
3   }
4   ...
5 - if(endIndex<0){
6 + if((endIndex<0)||(endIndex<startIndex)){
```

Listing 2: Patch of Chart-9 in Defects4J benchmark.

From the patch we can see that the root cause is the condition of the second `if` statement in line 5, where an additional condition expression `endIndex<startIndex` should be inserted as shown in line 6. However, after calculating the suspiciousness scores of predicates from the **branches** group, we will notice that the predicate `!(start==null)` taken from line 1 has higher suspiciousness than the predicate `endIndex<0` from line 5, leading to an incorrect localization of line 1. That is, while generally the **branches** group plays positive role, it may also lead to negative effects.

Based on the observation, one potential direction to improve this kind of situation is developing a more effective approach for selecting predicates. Since the performance of predicates vary from location to location, the selection should also be conditional over the context of the predicates. Techniques such as machine learning or invariant generation may be used.

TABLE IV: Integration fault localization results with existing techniques on Defects4J benchmark.

	Level 2								Level 4							
	Top-1		Top-3		Top-5		Top-10		Top-1		Top-3		Top-5		Top-10	
	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+
Chart	11	**13**	19	**18**	21	**18**	23	**25**	13	**14**	19	**18**	21	**18**	24	**20**
Math	49	**58**	78	**81**	84	**87**	90	**94**	63	**67**	83	**86**	85	**91**	93	**95**
Lang	42	**45**	55	**55**	57	**56**	60	**60**	47	**51**	59	**57**	61	**61**	61	**61**
Time	12	**12**	15	**16**	17	**19**	18	**20**	10	**12**	12	**17**	16	**19**	19	**20**
Closure	30	**46**	47	**64**	52	**68**	59	**81**	35	**41**	57	**64**	64	**76**	74	**88**
Total	144	**174**	214	**234**	231	**248**	250	**280**	168	**185**	230	**242**	247	**265**	271	**284**
Percent	40.3%	**48.7%**	59.9%	**65.5%**	64.7%	**69.5%**	70.0%	**78.4%**	47.1%	**51.8%**	64.4%	**67.8%**	69.2%	**74.2%**	75.9%	**79.6%**

In the table, the columns leading by "-" denote the original result of the fault localization framework COMBINEFL, while the columns leading by "+" denote the result after integrating PREDFL.

B. Better Formulas

In our study, we found that applying SBFL formulas to SD predicates could achieve significantly better performance. However, SBFL formulas are designed for program elements, and thus may not be the best choice for SD predicates. For instance, SBFL formulas do not utilize the two values F_0 and S_0. Future studies could explore the design space of risk evaluation formulas, and check if better formulas could be discovered.

VIII. DISCUSSION

Threats to internal validity is related to errors and selection bias in the experiment. To mitigate this risk, two authors of the paper with at least five-year developing experience carefully checked the implementation of our experiment with code review. Moreover, we have also run our approach multiple times to further guarantee the correctness of implementation. Besides, the selection of metrics in our experiment may also affect the evaluation results and corresponding conclusions. To tackle this problem, we selected two distinct metrics in our study, i.e., Top-k recall and ***EXAM*** score, both of which are prevalently employed by previous research.

Threats to external validity is related to the generalizability of our approach. In our experiment, we selected a commonly used benchmark, Defects4J, as our experimental dataset, which contains faults from real-world projects related to diverse software projects. Moreover, we presented and discussed the final results against existing approaches to show the effectiveness of our combined approach. As studied by Just et al. [54] that developer-provided triggering test cases could potentially overestimate the performance of fault localization techniques. This paper inherits this threat from the dataset as well. However, the results of our study are still informative with multiple-dimension comparisons. Moreover, the unified model and the implications learned from the study are free from this threat.

IX. RELATED WORK

In this section, we discuss related work with respect to four aspects. First, we explain spectrum-based fault localization and statistical debugging techniques explored in this paper. Then we discuss recent fault localization approaches that combine different techniques to improve state-of-the-art. Finally, we discuss existing empirical analysis on fault localization.

Spectrum-based fault localization is one of the most prevalent fault localization techniques, which is broadly utilized in program debugging, such as automatic program repair [21], [23]. In the past decade, many SBFL approaches have been proposed. For example, Tarantula [2] is a representative approach early proposed by Jones and Harrold, then Abreu and colleagues did a more thorough study on it and proposed a superior formula, Ochiai [24], which is often used to measure similarity in the molecular biology domain. Recently, some other formulas were proposed [55]. Though many approaches exist, as introduced in Section II-A, different SBFL approaches follow the same paradigm that leverages program syntax coverage to compute suspicious scores for candidate program elements with a predefined formula. Recently, SBFL has been employed in different domains, such as in SQL predicates [56], model transformations [57], logic-based reasoning [58], compiler bug isolation [59], and software product line [60], where it performs effective. Besides, existing studies try to improve SBFL technique with other optimized information such as suspicious variables [61] and reduced test cases by delta-debugging [62], or incorporating techniques like data-augmented diagnosis [63] and Feedback-based Goodness Estimate [64]. In contrast, in this paper, we further explore the effectiveness of SBFL predicates and formulas by combining statistical debugging under a unified model. Like existing studies [16], we selected a set of representative formulas in our evaluation to mitigate the selection bias. We leave a more thorough comparison among different formulas to future work.

Statistical debugging was proposed for remote program sampling by Liblit et al. [4], [5]. In the past years, many approaches have been proposed devoting to improving statistical debugging. Arumuga Nainar et al. [10] proposed to combine simple predicates to complex ones in order to improve the distinguishing ability of them. Zheng et al. [12] proposed to use a machine learning model to isolate predicates to clusters, each of which ideally can capture one single fault in a program that contains multiple faults. Similarly, Jiang and Su [65] proposed to cluster selected predicates to reconstruct candidate paths that are most likely to be faulty. Liu et al. [7], [13] proposed a new formula to compute the scores of predi-

cates, which distinguishes the predicate coverage distributions among failed and passed tests. Arumuga Nainar and Liblit [6] proposed to reduce the performance overhead of statistical debugging with adaptive binary instrumentation along a set of heuristics, and Zuo et. al [66] improved the performance of statistical debugging with abstraction refinement. Recently, Chilimbi and colleagues [11] proposed to improve traditional statistical debugging with path profiling, while Yu et al. [67] utilize predicates to reduce test cases. In this paper, we further explore the effectiveness of different predicates of SBFL and statistical debugging and investigate their different combinations aiming to boost current fault localization techniques, which is orthogonal to existing studies.

Fault localization combination denotes exploring different combination methods among fault localization techniques. Recently, many approaches have been proposed and improved the state-of-the-art. Santelices et al. [68] explored three kinds of coverage information, statement, branch and define-use pair coverage, where they found that the combination of three kinds of coverage can improve individual techniques. Recently, Tu et al. [69] proposed to combine SBFL with slicing-hitting-set-computation, which improves the effectiveness of SBFL. Xuan and Monperrus [38] proposed to leverage a learning-to-rank method to combine dozens of SBFL formulas, which obtained significant improvement against single formula. Similarly, Lucia et al. [70] proposed to fuse the diversity of existing spectrum-based fault localization techniques with considering dozens of variants of their combinations. Wang et al. [71] proposed to employ genetic algorithm and simulated annealing to look for optimal solutions for the combination of different spectrum-based fault localization techniques. Both of them were evaluated to outperform individual approaches. Recently several approaches have been proposed to leverage a learning-to-rank model to combine SBFL and supplementary information, such as invariants mined from executions [25], bug reports [72], code changes [18] and mutant coverage [17], [19], and improved existing techniques. Additionally, Li et al. [73] leveraged deep learning models to combine various features from the fault localization, defect prediction and information retrieval areas to further improve the state-of-the-art techniques. In this paper, we proposed a unified model of SBFL and statistical debugging, based on which we first systematically explored four different variations under the model. As a result, we found the effective configuration for each individual variation and proposed a simple combining method that incorporates their merits, which is evaluated to be effective and efficient. Moreover, our approach is orthogonal to existing techniques and evaluated to have the potential to further improve the state-of-the-art.

Empirical analysis on the performance of fault localization mainly focuses on analyzing existing fault localization techniques and investigating the essence of effectiveness. Harrold et al. [74] studied the relationships between program spectra and faults, which established the foundation of SBFL. Recently. Xie et al. [8] and Chen et al. [75] have theoretically revisited different formulas of SBFL approaches and found that there dose not exist the "best" formula that significantly outperforms the others. Pearson et al. [16] systematically explored the effectiveness of SBFL approaches and mutation based fault localization on both real-word faults and artificial faults, where they found the artificial faults are not significantly useful for predicting which fault localization techniques perform best on real faults. In this paper, we explored four variations that contributed to the improvement of combining different predicates, based on which we proposed a simple and effective fault localization approach. Moreover, our analysis provide several implications for future fault localization studies.

Empirical analysis on the usefulness of fault localization investigates the usability of automated fault localization techniques in practice. Recently, Parnin and Orso [76] and Xie et al. [77] studied the usefulness of automated fault localization techniques and observed that improvements in fault localization accuracy may not translate to improved manual debugging directly and even weaken programmers abilities in fault detection. However, a more rigorous study on larger-scale, real-world bugs performed by Xia et al. [78] reported that automated fault localization can positively impact debugging success and efficiency. Moreover, they found that the more accurate the fault localization results are, the more efficient the debugging process will be, demonstrating the practicability of automated fault localization techniques in practice. Similarly, Debroy and Wong [79] empirically evaluated that better fault localization can potentially improve the effectiveness of fault-fixing processes as well. As a consequence, in this paper, we target to improve the accuracy of current fault localization techniques and explore the combination of spectrum-based fault localization and statistical debugging.

X. Conclusion

In this paper, we empirically investigated different combinations of spectrum-based fault localization and statistical debugging techniques, for which we proposed a unified model. In addition, we systematically explored four variations under the model, which leads to several findings: (1) among all predicates, those from existing conditions contribute most to the Top-1 fault localization accuracy; (2) fine-grained data collection contributes more effective fault localization with little more execution overhead; (3) a linear combination of suspicious scores from SBFL and SD predicates leads to the best result. Based on our findings, we proposed a new fault localization technique, PredFL, which is evaluated to be complementary to existing techniques as it could further improve state-of-the-art by combining with existing techniques.

Acknowledgment

This work was partially supported by the National Key Research and Development Program of China under Grant No.2017YFB1001803, and National Natural Science Foundation of China under Grant Nos. 61672045, 61529201 and 6167254.

REFERENCES

[1] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*, 2002.

[2] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*. New York, NY, USA: ACM, 2005, pp. 273–282.

[3] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*, 2007, pp. 89–98.

[4] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*. New York, NY, USA: ACM, 2003, pp. 141–154.

[5] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI*. New York, NY, USA: ACM, 2005, pp. 15–26.

[6] P. Arumuga Nainar and B. Liblit, "Adaptive bug isolation," ser. ICSE, New York, NY, USA, 2010, pp. 255–264.

[7] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," ser. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 286–295.

[8] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, 2013.

[9] T. D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *2013 IEEE International Conference on Software Maintenance*, 2013, pp. 380–383.

[10] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit, "Statistical debugging using compound boolean predicates," ser. ISSTA, New York, NY, USA, 2007, pp. 5–15.

[11] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," ser. ICSE, Washington, DC, USA, 2009, pp. 34–44.

[12] A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: Simultaneous identification of multiple bugs," ser. ICML, New York, NY, USA, 2006, pp. 1105–1112.

[13] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *TSE*, vol. 32, no. 10, pp. 831–848, Oct 2006.

[14] L. Jiang and Z. Su, "Context-aware statistical debugging: From bug predictors to faulty control flow paths," ser. ASE, New York, NY, USA, 2007, pp. 184–193.

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014, pp. 437–440.

[16] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," ser. ICSE '17, 2017, pp. 609–620.

[17] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," pp. 92:1–92:30, 2017.

[18] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," ser. ISSTA, New York, NY, USA, 2017, pp. 273–283.

[19] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[20] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. Lamelas, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, 2017.

[21] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *ICSE*, 2017.

[22] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *SANER*, 2016, pp. 213–224.

[23] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," ser. ASE, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155644

[24] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," ser. PRDC. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46.

[25] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*, 2016, pp. 177–188.

[26] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the Defects4J dataset," *Empirical Software Engineering*, pp. 1–29, 2016.

[27] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *ASE*, 2017.

[28] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *ASE*. IEEE Press, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3155562.3155643

[29] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *ICSE*, 2018.

[30] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *ISSTA*, 2018.

[31] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *ICSE*, 2018.

[32] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA*. New York, NY, USA: ACM, 2019, pp. 19–30.

[33] J. Jiang, Y. Xiong, and X. Xia, "A manual inspection of defects4j bugs and its implications for automatic program repair," *Science China Information Sciences*, vol. 62, p. 200102, Sep 2019.

[34] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," ser. ISSTA, New York, NY, USA, 2016, pp. 165–176.

[35] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *ASE*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99.

[36] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, no. 1, pp. 290–308, March 2014.

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

[38] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *ICSME*, 2014, pp. 191–200.

[39] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *ISSTA*, 2013, pp. 314–324.

[40] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *TSE*, pp. 707–740, 2016.

[41] W. E. Wong, V. Debroy, R. Golden, X. Xu, and B. Thuraisingham, "Effective software fault localization using an rbf neural network," *IEEE Transactions on Reliability*, pp. 149–169, 2012.

[42] E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *2008 1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 42–51.

[43] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, "A deeper look into bug fixes: Patterns, replacements, deletions, and additions," in *MSR*. New York, NY, USA: ACM, 2016, pp. 512–515.

[44] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006, pp. 272–281.

[45] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*, March 2014, pp. 153–162.

[46] X. Zhang, N. Gupta, and R. Gupta, "Pruning dynamic slices with confidence," ser. PLDI, New York, NY, USA, 2006, pp. 169–180.

[47] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Softw. Test. Verif. Reliab.*, pp. 605–628, Aug. 2015.

[48] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections: Hit or miss?" ser. ESEC/FSE '11, New York, NY, USA, 2011, pp. 322–331.

[49] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, June 2012, pp. 14–24.

[50] A. Schroter, A. Schrter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *MSR 2010*, May 2010, pp. 118–121.

[51] C. Hammacher, "Design and implementation of an efficient dynamic slicer for Java," Bachelor's Thesis, Nov. 2008.

[52] T. Wang and A. Roychoudhury, "Using compressed bytecode traces for slicing java programs," in *ICSE*, May 2004, pp. 512–521.

[53] H. Pan and E. H. Spafford, "Heuristics for automatic localization of software faults," *World Wide Web*, 1992.

[54] R. Just, C. Parnin, I. Drosos, and M. D. Ernst, "Comparing developer-provided to user-provided tests for fault localization and automated program repair," in *ISSTA*, 2018, pp. 287–297.

[55] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 11:1–11:32, 2011.

[56] Y. Guo, N. Li, J. Offutt, and A. Motro, "Exoneration-based fault localization for sql predicates," *Journal of Systems and Software*, vol. 147, pp. 230 – 245, 2019.

[57] J. Troya, S. Segura, J. A. Parejo, and A. Ruiz-Cortés, "Spectrum-based fault localization in model transformations," *ACM Trans. Softw. Eng. Methodol.*, pp. 13:1–13:50, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3241744

[58] I. Pill and F. Wotawa, "Spectrum-based fault localization for logic-based reasoning," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2018, pp. 192–199.

[59] J. Chen, J. Han, P. Sun, L. Zhang, D. Hao, and L. Zhang, "Compiler bug isolation via effective witness test program generation," in *ESEC/FSE*. New York, NY, USA: ACM, 2019, pp. 223–234.

[60] A. Arrieta, S. Segura, U. Markiegi, G. Sagardui, and L. Etxeberria, "Spectrum-based fault localization in software product lines," *Information and Software Technology*, vol. 100, pp. 18 – 31, 2018.

[61] J. Kim, J. Kim, and E. Lee, "Vfl: Variable-based fault localization," *Information and Software Technology*, vol. 107, pp. 179 – 191, 2019.

[62] A. Christi, M. L. Olson, M. A. Alipour, and A. Groce, "Reduce before you localize: Delta-debugging and spectrum-based fault localization," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct 2018, pp. 184–191.

[63] A. Elmishali, R. Stern, and M. Kalech, "Data-augmented software diagnosis," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, pp. 4003–4009. [Online]. Available: http://dl.acm.org/citation.cfm?id=3016387.3016470

[64] N. Cardoso and R. Abreu, "A kernel density estimate-based approach to component goodness modeling," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, ser. AAAI'13. AAAI Press, 2013, pp. 152–158. [Online]. Available: http://dl.acm.org/citation.cfm?id=2891460.2891482

[65] L. Jiang and Z. Su, "Context-aware statistical debugging: From bug predictors to faulty control flow paths," ser. ASE, New York, NY, USA, 2007, pp. 184–193.

[66] Z. Zuo, L. Fang, S.-C. Khoo, G. Xu, and S. Lu, "Low-overhead and fully automated statistical debugging with abstraction refinement," ser. OOPSLA. New York, NY, USA: ACM, 2016, pp. 881–896. [Online]. Available: http://doi.acm.org/10.1145/2983990.2984005

[67] Y. Yu, J. Jones, and M. J. Harrold, "An empirical study of the effects of test-suite reduction on fault localization," in *ICSE*, 2008, pp. 201–210.

[68] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," ser. ICSE. Washington, DC, USA: IEEE Computer Society, 2009, pp. 56–66.

[69] J. Tu, X. Xie, T. Y. Chen, and B. Xu, "On the analysis of spectrum based fault localization using hitting sets," *Journal of Systems and Software*, vol. 147, pp. 106 – 123, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218302231

[70] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *ASE*. New York, NY, USA: ACM, 2014, pp. 127–138. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642983

[71] Shaowei Wang, D. Lo, L. Jiang, Lucia, and H. C. Lau, "Search-based fault localization," in *ASE 2011*, Nov 2011, pp. 556–559.

[72] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: Better together," in *ESEC/FSE*. New York, NY, USA: ACM, 2015, pp. 579–590.

[73] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *ISSTA*. New York, NY, USA: ACM, 2019, pp. 169–180.

[74] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, pp. 171–194.

[75] T. Y. Chen, X. Xie, F. C. Kuo, and B. Xu, "A revisit of a theoretical analysis on spectrum-based fault localization," in *2015 IEEE 39th Annual Computer Software and Applications Conference*, July 2015, pp. 17–22.

[76] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011, pp. 199–209.

[77] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of automatic debugging via human focus-tracking analysis," in *ICSE*. ACM, 2016, pp. 808–819.

[78] 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," *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 267–278, 2016.

[79] V. Debroy and W. E. Wong, "Combining mutation and fault localization for automated program debugging," *Journal of Systems and Software*, vol. 90, pp. 45 – 60, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121213002616