Influential Global and Local Contexts Guided Trace Representation for Fault Localization

ZHUO ZHANG*, School of Big Data & Software Engineering, Chongqing University, China
YAN LEI†, School of Big Data & Software Engineering, Chongqing University, China
TING SU, Software Engineering Institute, East China Normal University, China
MENG YAN, School of Big Data & Software Engineering, Chongqing University, China
XIAOGUANG MAO, College of Computer, National University of Defense Technology, China
YUE YU, College of Computer, National University of Defense Technology, China

Trace data is critical for fault localization (FL) to analyze suspicious statements potentially responsible for a failure. However, existing trace representation meets its bottleneck mainly in two aspects: (1) the trace information of a statement is restricted to a local context (*i.e.,* a test case) without the consideration of a global context (*i.e.,* all test cases of a test suite); (2) it just uses the 'occurrence' for representation without strong FL semantics.

Thus, we propose **UNITE**: an infl**U**ential co**N**text-Gu**I**ded **T**race r**E**presentation, representing the trace from both global and local contexts with influential semantics for FL. UNITE embodies and implements two key ideas: (1) UNITE leverages the widely-used weighting capability from local and global contexts of information retrieval to reflect how important a statement (a word) is to a test case (a document) in all test cases of a test suite (a collection), where a test case (a document) and all test cases of a test suite (a collection) represent local and global contexts respectively; (2) UNITE further elaborates the trace representation from 'occurrence' (weak semantics) to 'influence' (strong semantics) by combing program dependencies. The large-scale experiments on 12 FL techniques and 22 programs show that UNITE significantly improves FL effectiveness.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: fault localization; trace representation; statement weighting; program dependence; suspiciousness;

ACM Reference Format:
Zhuo Zhang, Yan Lei, Ting Su, Meng Yan, Xiaoguang Mao, and Yue Yu. 2022. Influential Global and Local Contexts Guided Trace Representation for Fault Localization. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (October 2022), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*Also with Guangzhou College of Commerce, Guangzhou, China.
†Corresponding author and also with Peng Cheng Laboratory, ShenZhen, China.

Authors' addresses: Zhuo Zhang, School of Big Data & Software Engineering, Chongqing University, Chongqing, China, zz8477@126.com; Yan Lei, School of Big Data & Software Engineering, Chongqing University, Chongqing, China, yanlei@cqu.edu.cn; Ting Su, Software Engineering Institute, East China Normal University, Shanghai, China, tsu@sei.ecnu.edu.cn; Meng Yan, School of Big Data & Software Engineering, Chongqing University, Chongqing, China, mengy@cqu.edu.cn; Xiaoguang Mao, College of Computer, National University of Defense Technology, Changsha, China, xgmao@nudt.edu.cn; Yue Yu, College of Computer, National University of Defense Technology, Changsha, China, yuyue@nudt.edu.cn.

1 INTRODUCTION

In software development and maintenance, debugging is one of the most expensive and time-consuming processes [18, 38, 48]. To reduce the cost, researchers have developed many fault localization (FL) techniques to provide automated assistance in seeking the faults that cause a failure [13, 23, 40, 52, 55, 57, 69].

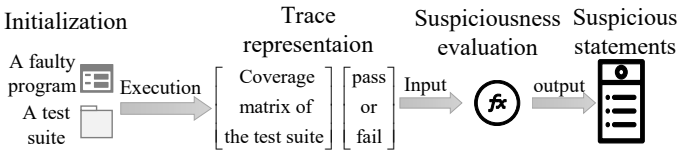

Fig. 1. The typical process of FL.

Fig. 1 shows the typical process of FL. Suppose that we have a faulty program and a test suite for initialization. Then, FL executes the test suite on the program to collect and abstract the execution traces as a coverage matrix for trace representation, where an element denotes a statement *covered* (*i.e.,* the value of '1' denoting *occurrence*) or *not covered* (*i.e.,* the value of '0' representing *non-occurrence*) by a specific test case. Trace representation also constructs an error vector to represent the test results (*i.e.,* '1' for *fail* and '0' for *pass*). Next, FL takes as input the trace representation, and uses an evaluation model (*e.g.,* correlation coefficients [15, 16, 35, 39] and neural networks [53, 56, 66–68]) to evaluate the suspiciousness of each statement of being faulty. Finally, FL outputs the suspicious statements as a ranking list of all statements in descending order of suspiciousness.

Although trace representation is an indispensable component of the FL process, it still has some limitations. Existing trace representation uses the binary state of a statement (*i.e., occurrence* or *non-occurrence*) in a test case, which is restricted to a local context (*i.e.,* a test case) without the consideration of a global context (*i.e.,* all test cases of a test suite). For example, suppose that we have two statements s_1 and s_2, where s_1 is only executed by the test case t_1 and s_2 is not only executed by t_1 but also executed by many other test cases. Considering the global context of a test suite, s_1 should be more important than s_2 for t_1 since s_1 only occurs in t_1. For another example, suppose that we have two test cases t'_1 and t'_2, where t'_1 executes 10 statements including the statement s'_1 and t'_2 covers 100 statements including s'_1. Based on the global context of a test suite, s'_1 should be more important for t'_1 in comparison to t'_2 since t'_1 executes less statements than t'_2. However, existing trace representation using binary state of a statement cannot capture such importance information. Therefore, its information is limited, *e.g.,* it cannot show to what degree of the importance of a statement is in an execution. Even if some approaches [11, 45] seek to enrich the representation from the local context of a test case itself, the lack of global context of all test cases of a test suite can cause some biases posing a negative effect on the effectiveness of fault localization [22], *i.e.,* their representation actually performs worse than the widely-used binary trace representation [22]. Furthermore, existing trace representations mainly use the 'occurrence' semantics whereas the occurrence of a statement in a test case does not necessarily mean that the execution of the statement influences the program output. For example, suppose that a failing test case t_f executes two statements s_{f1} and s_{f2}, where the variable causing the faulty output of t_f is only computed by s_{f1}. In this case, we should exclude s_{f2} since its execution does not influence the faulty output. Nevertheless, the existing trace representation using statement coverage cannot capture such influence information. Thus, it lacks a strong FL semantics, restricting a deep analysis of suspicious evaluation model in evaluating the suspiciousness of a statement of being faulty.

Therefore, this paper proposes **UNITE**: an inflUential coNtext-GuIded Trace rEpresentation for effective FL, exploiting global and local contexts guided trace representation with influential semantics. Similar to the coverage matrix in Fig. 1, UNITE abstracts trace representation as a matrix by redefining the element which combines global and local contexts with influential semantics. Inspired by the widely-used word weighting capability from both local and global contexts of term frequency-inverse document frequency [41, 44] in information retrieval, UNITE applies this promising capability on trace representation in fault localization. Based on the term frequency-inverse document frequency, the idea of UNITE embodying the global and local contexts is that (1) if a statement is executed by many test cases, its weight should be lower for these test cases since it is difficult to distinguish the statement in these test cases; (2) if a statement is executed by a few test cases and the executed statements in these test cases have a small size, its weight should be higher for these test cases since the statement is more important to these test cases. With the weights of a statement in all test case of a test suite, UNITE can build linkages between the statement and the test results (*i.e.,* passing or failing) of test cases. To realize this idea, first, UNITE reformulates the trace information of a statement as the weight of a word in information retrieval. For an analogy, UNITE uses the three sources of information for FL: a statement (a word), a test case (a document) and all test cases of a test suite (a collection). UNITE defines the trace representation of a statement as the weight of the statement (the word) by increasing proportionally to the number of times a statement (the word) occurs in the test case (the document), and being offset by the number of the test cases (the documents) in all test cases of a test suite (the collection) that contain the statement (the word), which helps to adjust for the fact that some statements (words) occur more frequently in general. Thus, UNITE elaborates **the local context** as **term frequency** that increases proportionally to the number of times a statement occurs in the test case (*i.e.,* **statement frequency**), and **the global context** as **inverse document frequency** that is offset by the number of the test cases in all test cases of a test suite that contain the statement (*i.e.,* **inverse test case frequency**).

Thus, UNITE elaborates **the local context** as **term frequency** that increases proportionally to the frequency of a statement occurs in the test case (*i.e.,* **statement frequency**), and **the global context** as **inverse document frequency** that is offset by the number of the test cases in all test cases of a test suite that contain the statement (*i.e.,* **inverse test case frequency**). Although UNITE considers trace representation from both local and global contexts, it still relies on the occurrence frequency of a statement or a test case. Therefore, UNITE further combines program dependencies into trace representation for upgrading the FL semantics. Specifically, UNITE uses program slicing [2, 47, 61] to identify those statements whose execution influences the incorrect output according to program dependencies. Then, UNITE updates higher weights of those statements with influential semantics, and thus upgrades the '**occurrence**' (weak semantics) into the '**influence**' (strong semantics).

Since UNITE follows the widely-used matrix structure of most FL techniques, it means that UNITE may serve as a universal representation for most FL techniques. To evaluate the potential and the effectiveness of UNITE, we apply UNITE to 12 state-of-the-art FL techniques (*e.g.,* Dstar [54], CNN-FL [66], ProFL [28] and DeepRL4FL [26]) and conduct large-scale experiments on 22 benchmark programs. The results show that UNITE significantly improves FL effectiveness, *e.g.,* the average improvement for the important Top-N metric [17], *i.e., Top-1*, *Top-3*, *Top-5* and *Top-10*, increases up to 2.19%, 11.18%, 11.47% and 14.18%, respectively.

The main contributions of this paper can be summarized as:

- We propose UNITE: an influential context-guided trace representation for FL by combining global and local contexts with influential semantics.

- We demonstrate the potential of UNITE as a universal representation for a wide spectrum of the state-of-the-art FL techniques.
- We evaluate the effectiveness of UNITE across various 22 real-life large programs, showing that the UNITE is effective at improving FL.
- We open source the replication package online[1], including the source code, datasets and running examples.

The structure of the rest paper is organized as follows. Section 2 introduces related work. Section 3 depicts our approach UNITE. Section 4 and Section 5 present our large-scale experiments and the discussion. And Section 6 concludes the whole study and mentions future work.

2 RELATED WORK

This section surveys closely related work on fault localization (FL) from its two parts: **trace representation** and **suspiciousness evaluation**. More other work can be found in the survey [55].

2.1 Trace Representation

$$\begin{matrix} & \textit{N statements} & & & & \textit{errors} \\ \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1N} \\ x_{21} & x_{22} & \dots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \dots & x_{MN} \end{bmatrix} & & & & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \end{matrix}$$

Fig. 2. FL Trace Representation on M test cases of a test suite.

FL usually defines a matrix (*i.e.,* a coverage matrix and an error vector) to represent the trace of each statement in each test case of a test suite and their corresponding test results. Next, FL takes as input the trace representation for its suspiciousness evaluation. The trace representation records the runtime information and test results of a test suite including the execution information of statements.

Fig. 2 shows the definition of the FL trace representation (*i.e.,* a $M \times (N+1)$ matrix). Specifically, given a program P with N statements (s_1, s_2, ..., s_N), it is executed by a test suite T with M test cases (t_1, t_2, ..., t_M), which contain at least one failing test case (see Fig. 2). The element x_{ij}=1 means that the statement s_j occurs in (*i.e.,* is covered by) the test case t_i, and x_{ij}=0 otherwise. The $M \times N$ matrix records the execution information of each statement in the test suite T. The error vector e represents the test results. The element e_i equals to 1 if the test case t_i failed, and 0 otherwise. The error vector shows the test results of each test case (*i.e.,* failure or non-failure).

Even if some research [11, 45, 65, 70] tries to enrich trace representation of FL by using other information (*e.g.,* statement frequency), these approaches like the binary representation still have some limitations: (1) the trace information of a statement is restricted to a local context (*i.e.,* a test case) without the consideration of a global context (*i.e.,* all test cases of a test suite); (2) they just uses the 'occurrence' for representation without strong FL semantics. Even worse, recent work [22] shows that these approaches (*e.g.,* [11, 45]) cause some bias posing a negative effect on fault localization effectiveness, *i.e.,* their elaboration on trace representation is not better than the binary state of FL trace representation.

[1] https://github.com/oy-sarah/UNITE

This motivates our work to solve the above two problems by proposing an FL trace representation to combine both local and global contexts and upgrade the 'occurrence' (weak semantics) into 'influence' (strong semantics).

2.2 Suspiciousness Evaluation

Based on the trace representation in Fig. 2, researchers develop many suspiciousness evaluation models to evaluate the suspiciousness of a statement of being faulty. We can roughly classify the suspiciousness evaluation models into two categories.

One category is suspiciousness evaluation using correlation coefficients, which are widely studied by the spectrum-based fault localization (SFL) researchers [15, 16, 35, 39]. Correlation coefficients are suspiciousness evaluation formulas, and SFL uses the trace representation to define four variables for the formulas as follows:

$$a_{np}(s_j) = \sum_{i \in np(s_j)} (1 - x_{ij}), \quad np(s_j) = \{i | (x_{ij} = 0) \wedge (e_i = 0)\} \tag{1}$$

$$a_{ep}(s_j) = \sum_{i \in ep(s_j)} x_{ij}, \quad ep(s_j) = \{i | (x_{ij} > 0) \wedge (e_i = 0)\} \tag{2}$$

$$a_{nf}(s_j) = \sum_{i \in nf(s_j)} (1 - x_{ij}), \quad nf(s_j) = \{i | (x_{ij} = 0) \wedge (e_i = 1)\} \tag{3}$$

$$a_{ef}(s_j) = \sum_{i \in ef(s_j)} x_{ij}, \quad ef(s_j) = \{i | (x_{ij} > 0) \wedge (e_i = 1)\} \tag{4}$$

Eq. (1), Eq. (2), Eq. (3) and Eq. (4) show the computation of a_{np}, a_{nf}, a_{ep}, and a_{ef} for the statement j (*i.e.,* s_j), denoting the number of passing/failing test cases in which the statement was/wasn't executed. Based on the four variables for each statement (*i.e.,* a_{np}, a_{nf}, a_{ep}, and a_{ef}), SFL defines many suspiciousness evaluation formulas using correlation coefficients to evaluate the suspiciousness of each statement being faulty. Researchers have conducted both theoretical [59, 60] and empirical [39] analysis on finding the optimal SFL formulas using correlation coefficients, and identified seven effective ones, namely ER1', ER5, GP02, GP03, GP19, Ochiai and Dstar. Table 1 shows all the seven effective suspiciousness evaluation formulas using correlation coefficients[2]. Based on these formulas, some researchers incorporate more useful information into suspiciousness evaluation, *e.g.,* the popular and promising approach ProFL [28] leverages repair information as feedback.

The other one category is suspiciousness evaluation using neural networks, which are recently studied by the deep learning-based fault localization (DLFL) researchers [25, 46, 67, 72]. Based on the trace representation, this category tries to utilize artificial neural network with hidden layers [9, 20, 24, 34, 49, 68] to learn a fault localization model reflecting the statistical coincidences between test results (*i.e.,* failing or passing) and the executions of the different statements of a program (*i.e.,* occurrence or non-occurrence). We will introduce four representative suspiciousness evaluation models used in our experiments, namely MLP-FL [71], CNN-FL [66], BiLSTM-FL [68] and DeepRL4FL [26].

Fig. 3 shows the architecture of suspiciousness evaluation of DLFL using neural networks: one input layer, deep learning components with several hidden connected layers, and one output layer. In the input layer, the coverage matrix and the error vector of FL trace representation in the Fig. 2 are used as the training samples and their corresponding

[2]The * in D* formula is usually assigned to 2.

Table 1. Suspiciousness evaluation using correlation coefficients.

Name		Formulas	Name	Formulas
ER1'	Naish1	$\begin{cases} -1 & if\ a_{ne} > 0 \\ a_{np} & if\ a_{ne} \leq 0 \end{cases}$	GP02	$2\left(a_{ef} + \sqrt{a_{np}}\right) + \sqrt{a_{ep}}$
	Optimal_P	$a_{ef} - \frac{a_{ep}}{a_{ep}+a_{np}+1}$	GP03	$\sqrt{\left\|a_{ef}^2 - \sqrt{a_{ep}}\right\|}$
	GP13	$a_{ef}\left(1 + \frac{a_{ep}}{2a_{ep}+a_{ef}}\right)$	GP19	$a_{ef}\sqrt{\left\|a_{ep} - a_{ef} + a_{nf} - a_{np}\right\|}$
ER5	Wong1	a_{ef}	Dstar	$\frac{a_{ef}^*}{a_{nf}+a_{ep}}$
	Russel_Rao	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$		
	Binary	$\begin{cases} 0, & if\ a_{ne} > 0 \\ 1, & if\ a_{ne} \leq 0 \end{cases}$	Ochiai	$\frac{a_{ef}}{\sqrt{(a_{ef}+a_{nf})(a_{ef}+a_{ep})}}$

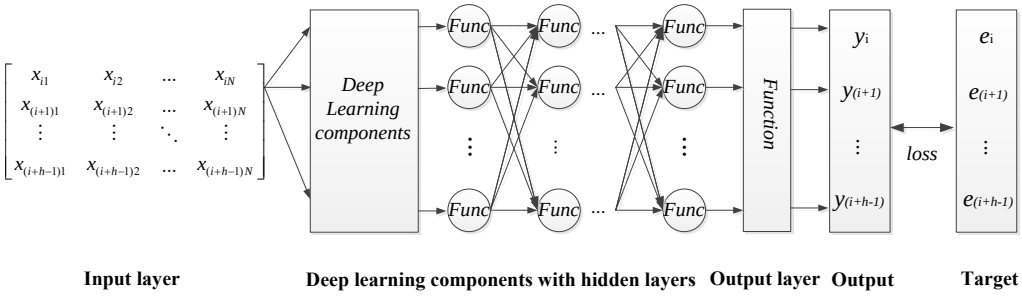

Fig. 3. Suspiciousness evaluation using neural networks.

labels, respectively. In other words, h rows of the matrix $M \times N$ and its corresponding error vector are used as an input, which are the coverage information of h test cases and their corresponding test results starting from the i-th row, where $i \in \{1,1+h,1+2h,...,1+(\lfloor M/h \rfloor+1)\times h\}$. In deep learning components with several hidden connected layers, MLP-FL, CNN-FL and BiLSTM-FL use multi-layer perceptron, convolutional neural network and bi-directional long short-term memory respectively. DeepRL4FL integrate these basic neural networks using multiple dimensions of features. In the output layer, DLFL uses *Sigmoid* function [24] because values sent into a *Sigmoid* function will be 0 to 1. Each element in the result vector of the *Sigmoid* function has difference with the corresponding element of the target vector. Back propagation algorithm is used to fine-tune the parameters of the model, and the goal is to minimize the difference between training result y and error vector e. The network is trained iteratively. Finally, DLFL using neural networks learns a trained model reflecting the relationship between statement coverage and test results. With the trained model, DLFL evaluates the suspiciousness of each statement.

Our work focuses on developing an effective universal representation for these suspciousness evaluation models, and can be widely used by these models.

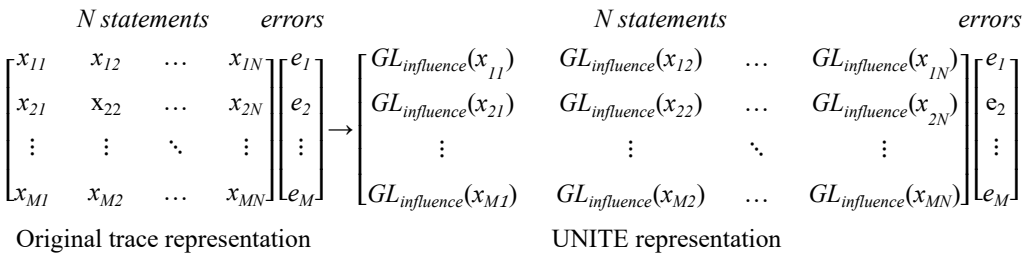

Fig. 4. The original FL trace representation and the UNITE representation of M test cases of a test suite.

3 APPROACH

3.1 Formulation

First, we should formulate the problem. Given a program P with N statements (s_1, s_2, ..., s_N), it is executed by M test cases T (t_1, t_2, ..., t_M). Fig. 4 shows the original FL trace representation and the UNITE representation. let us recall the original FL trace representation (see the left matrix of Fig. 4). x_{ij}=1 indicates that the statement s_j occurs in the test case t_i, and 0 otherwise. The error vector e represents the test results. The element e_i equals to 1 if the test case t_i failed, and 0 otherwise. Since the original FL trace representation serves as a universal input for most FL techniques, UNITE will keep its structure for wide FL applicability. Thus, as shown in Fig. 4, the core work of UNITE is to redefine the elements of the original FL trace representation with influential global and local contexts.

3.2 UNITE with Global and Local Contexts

In the field of information retrieval, term frequency-inverse document frequency (TF-IDF) [41] is a popular word weighting technique designed to reflect the importance of a word to a document (*i.e.,* local context) in a collection (*i.e.,* global context). Inspired by the TF-IDF, UNITE utilizes its promising weighting capability by elaborating the trace representation to reflect the importance of a statement to a test case (*i.e.,* local context) in all test cases of a test suite (*i.e.,* global context). Thus, for an analogy, the basic idea of UNITE with global and local contexts can be roughly summarized as that if a statement (keyword) occurs only in a few test cases (documents), then it is easy to lock the FL target (search target), and the weight of the statement (word) should be relatively large. If a statement (word) exists in a large number of test cases(documents), then it is not clear to find the goal with the statement (word), and the weight of the statement (word) should be small. As a reminder, a statement occurring in a test case means that a statement is covered by the test case. Since UNITE considers the times of a statement occurs in (is covered by) a test case, the specific content (*e.g.,* whitespace and variable names) of a statement itself will not affect UNITE.

To realize the above idea, UNITE defines **the local context** as the **term frequency** which increases proportionally to the frequency of a statement occurs in the test case (*i.e.,* **statement frequency**); and **the global context** as **inverse document frequency** which is offset by the number of the test cases in all test cases of a test suite that contain the statement (*i.e.,* **inverse test case frequency**). Specifically, UNITE defines the following TF_{local}, IDF_{global}, GL_{occur} as the local context, the global context and the combination of the global and local contexts, respectively.

$$TF_{local}\left(x_{ij}\right) = x_{ij} * \frac{1}{1+log(N(t_i))} \tag{5}$$

$$IDF_{global}\left(x_{ij}\right) = log(\frac{M}{DF(s_j)}) \tag{6}$$

$$GL_{occur}\left(x_{ij}\right) = TF_{local}\left(x_{ij}\right) * IDF_{global}\left(x_{ij}\right) \tag{7}$$

Based on the x_{ij} of the original FL trace representation (*i.e.,* the binary value of 1 or 0 of s_j in test case t_i), Eq. (5) calculates $TF_{local}(x_{ij})$, denoting the TF value of the statement s_j in the test case t_i (*i.e.,* **statement frequency** of s_j in test case t_i), where $N(t_i)$ means the number of executed statements in the test case t_i. Eq. (6) calculates $IDF_{global}(x_{ij})$, representing the IDF value of the statement s_j in the test suite (*i.e.,* **inverse document frequency** of s_j in the whole test suite), where $DF(s_j)$ indicates the number of test cases executing the statement s_j. In Eq. (5) and Eq. (6), we adopt the widely-used *log* function in TF-IDF. Eq. (7) calculates $GL_{occur}(x_{ij})$ via the the multiplication of $TF_{local}(x_{ij})$ and $IDF_{global}(x_{ij})$, denoting the TF-IDF value of the statement s_j in the test case t_i

Based on the Eq. (7), UNITE redefines an element of x_{ij} as $GL_{occur}(x_{ij})$ by combing the global and local contexts into trace representation.

3.3 UNITE with Influence Semantics

We can observe that the global and local contexts are constructed from statement coverage information. Although the statement coverage information is useful and effective, an occurrence of a statement in a test case does not necessarily mean that the execution of the statement will influence the output of the test case. For example, for a statement stm, its execution does not influence the incorrect output. Even if the statement stm has a high value of GL_{occur}, the statement stm should have the lowest weight because its execution is independent of the incorrect output. Thus, the new trace representation (*i.e.,* $GL_{occur}(x_{ij})$) still relies on the 'occurrence' semantics (*i.e.,* the occurrence frequency of a statement or a test case), and thus cannot capture such 'influence' semantics (*i.e.,* whether the execution of a statement influences the output or not). To further improve FL effectiveness, this motivates us to integrate 'influence' semantics into the trace representation via using program slicing [2, 61] to capture whether the execution of a statement influences the output or not. Therefore, UNITE uses program slicing [2, 61] to elaborate the trace representation by upgrading the '**occurrence**' into the '**influence**' semantics.

Program slicing [2, 61] extracts the data and/or control dependencies of program statements to identify a subset of statements whose execution affects the output. It names the subset of statements as a slice. A slice is a program dependency graph showing how those statements influence the output according to data and/or control dependencies. Therefore, UNITE uses dynamic slicing [2, 61] on the output statement whose output value is incorrect to identify those statements affecting the faulty output value as an influential slice. Thus, an influential slice is defined as follows:

> **An influential slice:** statements that directly or indirectly affect the computation of the faulty output value of a failure through chains of dynamic data and/or control dependencies.

For the computation of an influential slice, we use the following slicing criterion.

$$influentialSC = (outStm, incorrectVar, failEx) \tag{8}$$

Where, $outStm$ is an output statement whose value of a variable (*i.e.,* $incorrectVar$) is incorrect in the execution of a failing test case (*i.e.,* $failEx$). Dynamic slicing collects runtime information along the execution path of a test case, *i.e.,* the set of the executed statements of a test case. It means that a test case with a smaller set of executed statements is usually easier for a dynamic slicing tool to perform efficient instrumentation and produce compressed traces for space optimization. Thus, for multiple failing test cases, the one with the least executed statements usually is beneficial for the efficiency of constructing an influential slice. From the efficiency aspect, UNITE chooses the failing test case having

Program P (maximal value of a,b,c)											Bug information
s_1: *Read*(a); s_2: *Read*(b); s_3: *Read*(c) ; s_4: *if*(c>a)*and* (c>b){ s_5: *max* = c;} s_6: *else if* (a<b){								s_7: *max* = a; s_8: *else* {*max* = b;}		The influential slice with t_5: {s_1, s_2, s_3, s_4, s_5, s_6}					s_6 is faulty. Correct form: *else if* (a>b){				
T	a,b,c	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	T	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	R
t_1	1,2,3	1	1	1	1	1	0	0	0	t_1	0	0	0	0	0	0	0	0	0
t_2	-2,-7,5	1	1	1	1	1	0	0	0	t_2	0	0	0	0	0	0	0	0	0
t_3	5,-6,-8	1	1	1	1	0	1	0	1	t_3	0	0	0	0	0.06	0	0	0	1
t_4	5,4,3	1	1	1	1	0	1	0	1	t_4	0	0	0	0	0.06	0	0	0	1
t_5	4,7,1	1	1	1	1	0	1	1	0	t_5	0	0	0	0	0.06	0.17	0	0	1
t_6	-1,2,1	1	1	1	1	0	1	1	0	t_6	0	0	0	0	0.06	0.17	0	0	1
MLP-FL	value	0.62	0.64	0.61	0.69	0.57	0.59	0.60	0.58	MLP-FL (UNITE)	0.56	0.28	0.19	0.46	0.79	0.95	0.97	0.88	
	rank	3	2	4	1	8	**6**	5	7		5	7	8	6	4	**2**	1	3	
ER5	value	0.67	0.67	0.67	0.67	0	0.67	0.33	0.33	ER5 (UNITE)	0	0	0	0	0	0.11	0.08	0	
	rank	1	2	3	4	8	**5**	6	7		6	7	8	4	5	**1**	2	3	

Fig. 5. An Example illustrating our approach.

the least executed statements to construct a slicing criterion in the Eq. (8), and inputs this slicing criterion into program slicing technique to construct an influential slice.

Based on the influential slice, UNITE defines Eq. (9) to combine influential semantics into trace representation.

$$GL_{influence}\left(x_{ij}\right) = \mathrm{GL}_{\mathrm{occur}}\left(x_{ij}\right) * SLICE\left(x_{ij}\right) \tag{9}$$

Where, $SLICE(x_{ij}) = 1$ if the statement $s_j \in influentialSC$; and 0 otherwise. Eq. (9) assigns the lowest value to those statements not in the influential slice because their executions do not influence the faulty output.

Finally, as shown in Fig. 4, UNITE defines $GL_{influence}(x_{ij})$ to replace the original x_{ij}, and models a new trace representation (*i.e.,* a new matrix) with influential global and local contexts. FL techniques (*e.g.,* SFL and DLFL in Section 2) take as input the UNITE representation to analyze and evaluate the suspiciousness of a statement of being faulty.

3.4 An Illustrative Example

Fig. 5 shows an example illustrating how UNITE is applied. As shown in Fig. 5, we have a program P with a fault at the statement s_6, and its function is to calculate the maximal value of three variables. The left six cells below each statement represent whether the statement is covered by the test case (1 for covered and 0 otherwise), evaluated by the original trace representation. The right six cells represent the $GL_{influence}$ values of each statement in each test case, evaluated by UNITE (see Eq. (9)). The rightmost cells below R indicate whether the test case is failing or not (1 for failing and 0 otherwise). In this illustrative example, we choose MLP-FL and ER5 as the representative for DLFL and SFL, which are described in Section 2. MLP-FL(UNITE) and ER5(UNITE) mean that MLP-FL and ER5 use the UNITE trace representation. UNITE uses the failing test case $t5$ to calculate the influence slice. Here, we can observe that

UNITE representation is more concise and precise than binary representation by purifying uninfluenctial statements and showing a magnitude of the importance of a statement in a test case.

After acquiring the trace representation, FL techniques take as input the representation to analyze and evaluate the suspiciousness of each statement of being faulty. For example, MLP-FL(UNITE) takes as input UNITE representation and its concrete process is as follows: first, UNITE constructs the MLP model with the number of input layer nodes being eight, three hidden layers with the number of each one's nodes being 10, and the number of output layer nodes being 1; then, we input the vector t_1 (0,0,0,0,0,0,0,0) and its result 0, then vector t_2 (0,0,0,0,0,0,0,0) and its result 0 into the input layer until all the vectors of UNITE representation are all inputted into the network. After that, we train the network iteratively to acquire the relationship between the execution influence of a statement and the test results. Thirdly, we construct the virtual test set which is an eight dimensional unit matrix, then put it into the network, and finally obtain the suspiciousness values. Based on these information, MLP-FL(UNITE) outputs a ranking list of all statements in descending order. The original MLP-FL uses the binary representation to perform a similar process to evaluate the suspiciousness of each statement of being faulty. The results show that the faulty statement s_6 is ranked *2nd* by UNITE and ranked *6th* by the original MLP-FL.

Based on the binary representation and UNITE representation, ER5 and ER5(UNITE) both output a ranking list of all statements in descending order. The results show that the faulty statement s_6 is ranked *1st* by UNITE and ranked *5th* by the original ER5 using the binary representation. It should be noted that when the statements have the same suspiciousness value, we adopt the widely strategy by ranking them in the ascending order of their line numbers. As shown in Fig. 5, although s_1, s_2, s_3, s_4 and s_6 have the same highest suspiciousness value in ER5, s_6 is ranked 5th for its larger line number. Thus, for different strategies of breaking the tie, the ranks of those statements with the same suspiciousness value may be slightly different.

We can first observe that global and local contexts of UNITE work. Since the statements s_1, s_2, s_3 and s_4 are executed by all the 6 test cases, their GL_{occur} values are 0 in comparison to the other statements, and thus their $GL_{influence}$ are 0. Furthermore, the statement s_6 and s_7 acquire a decimal value, rather than a binary value, showing how important of a statement is in a test case. Then, the influence semantics of UNITE works. Due to using the influential slice, the execution of s_5 and s_8 do not influence the faulty output of $t5$, their *SLICE* values are 0 and thus their $GL_{influence}$ are 0 Thus, based on this illustrative example, we can observe that the two parts of UNITE (*i.e.,* global and local contexts, and influential semantics) both contribute to FL effectiveness, leading to better FL effectiveness over the original trace representation. Section 4.3.2 offers an evaluation on the contribution of each part of UNITE to FL effectiveness.

4 EXPERIMENTS

4.1 Experimental Setup

Benchmarks The experiments choose the subject programs for the two reasons: (1) they are the widely used large-sized programs (*e.g.,* [24, 33, 35–37, 39, 42, 55, 66, 67]) in fault localization; (2) they are easy to be acquired for enabling comparable and reproducible studies. Table 2 summarizes the 22 subject programs. For each program, it provides a brief functional description (column 'Description'), the number of faulty versions used (column 'Versions'), the number of thousand lines of statements (column 'KLOC'), the number of test cases (column 'Test') and the type of the faults (column 'Type'). The first four programs are real faults, among which *python*, *gzip* and *libtiff* are collected

Table 2. Subject programs.

Program	Description	Versions	KLOC	Test	Type
python	General-purpose language	8	407	355	Real
gzip	Data compression	5	491	12	Real
libtiff	Image processing	12	77	78	Real
space	ADL interpreter	35	6.1	13585	Real
spoon	Java code analysis & transformation	31	76	1114	Real
dubbo	Apache incubator dubbo	1	0.6	90	Real
jackson-databind	General data binding	13	99	1711	Real
oak	Apache jackrabbit oak	1	1.8	2403	Real
debezium	Platform for change data capture	4	53	508	Real
byte-buddy	Runtime code generation for the JVM	3	140	8066	Real
AutomatedCar	Passenger vehicle behavior simulator	1	2	48	Real
cash-count	Accounting software back-end	2	0.7	16	Real
nanoxml_v1	XML parser	7	5.4	206	Seeded
nanoxml_v2	XML parser	7	5.7	206	Seeded
nanoxml_v3	XML parser	10	8.4	206	Seeded
nanoxml_v5	XML parser	7	8.8	206	Seeded
chart	JFreeChart	26	96	2205	Real
math	Apache commons math	106	85	3602	Real
lang	Apache commons-lang	65	22	2245	Real
time	Joda-Time	27	53	4130	Real

from ManyBugs[3], and *space* is acquired from the SIR[4]. The next seven programs are real faults from BEARS[5]. Then, the next four programs are seeded faults of the four sperate releases of *nanoxml* acquired from the SIR. The last four programs (*i.e., chart*, *math*, *lang* and *time*) are acquired from Defectcs4J[6]. As a reminder, since the recent studies [10, 72] have identified over-fitting benchmarks (*e.g.,* Defects4J) for FL, we use the recently recommended benchmarks [29] (*e.g.,* BEARS) to alleviate this problem. Therefore, we do not include the experimental results of Defects4J in Section 4.3 and provide a discussion on the effect of benchmarks over-fitting on our approach using Defects4J in Section 5.1.

We use JSlice[7] and Javaslicer[8] for slicing Java programs, and WET[9] for slicing C Programs. Due to running environments, the tools cannot slice some faulty versions, and we remove these versions in our evaluation.

Baselines According to the extensive existing studies [26, 28, 31, 32, 39, 43, 51, 59, 60, 63, 66, 67], the experiments use the 12 state-of-the-art FL approaches as the baselines, *i.e.,* ER5, GP02, GP03, Dstar, ER1', GP19, Ochiai, MLP-FL, CNN-FL, BiLSTM-FL, ProFL and DeepRL4FL. We implement the 12 baselines including the parameters as described in their publications.

Environment The physical environment of the experiments is on a computer containing a CPU of Intel I5-2640 with 128G physical memory, and two 12G GPUs of NVIDIA TITAN X Pascal. The operating system is Ubuntu 16.04.3. We conducted the experiments on the MATLAB R2016b.

[3]ManyBugs, https://repairbenchmarks.cs.umass.edu/ManyBugs/.
[4]SIR, http://sir.unl.edu/portal/index.php.
[5]BEARS, https://github.com/bears-bugs/bears-benchmark.
[6]Defects4J, http://defects4j.org.
[7]http://jslice.sourceforge.net/.
[8]https://github.com/hammacher/javaslicer/.
[9]http://wet.cs.ucr.edu/.

4.2 Evaluation Metrics

We adopt four widely used metrics to evaluate the effectiveness of UNITE, namely *Top-N accuracy* [17, 38], *Mean Average Rank (MAR)* [24], *Mean First Rank (MFR)* [24] and Relative Improvement (*RImp*) [4, 7, 21]. A higher value of *Top-N Accuracy* means better localization effectiveness, while a lower value denotes better localization effectiveness for the other four metrics.

Top-N Accuracy It denotes the percentage of faults located within the first *N* position of a ranked list of all statements in descending order of suspiciousness returned by a FL approach.

Mean Average Rank (MAR) It is the mean of the average rank of all faults using a FL approach.

Mean First Rank (MFR) For a fault with multiple faulty statements, locating the first one is critical since the others may be located after that. *MFR* is the mean of the first faulty statement's rank of all faults using a localization approach.

Relative Improvement (RImp) It is to compare the total number of statements that need to be examined to find all faults using UNITE versus the number that need to be examined by without using UNITE.

4.3 Experimental Results

4.3.1 RQ1. **What is the FL effectiveness of UNITE compared with the original state-of-the-art FL baselines?**

We compare 12 state-of-the-art FL baselines using UNITE with the original ones to answer RQ1.

Table 3. *Top-N*, *MAR* and *MFR* comparison of 13 FL approaches using UNITE over without using UNITE

Comparison	top-1	top-3	top-5	top-10	MAR	MFR
ER5	1.09%	5.46%	9.29%	11.95%	421	263
ER5(UNITE)	**+1.64%**	**+9.84%**	**+8.20%**	**+9.91%**	**134**	**125**
GP02	1.09%	6.01%	8.20%	11.29%	464	289
GP02(UNITE)	**+2.19%**	**+8.74%**	**+11.47%**	**+11.66%**	**124**	**111**
GP03	1.09%	5.76%	10.14%	12.57%	417	251
GP03(UNITE)	**+1.64%**	**+11.18%**	**+7.11%**	**+7.66%**	**131**	**119**
Dstar	2.73%	6.56%	14.27%	23.50%	386	243
Dstar(UNITE)	**+0.55%**	**+5.94%**	**+3.76%**	**+4.92%**	**125**	**113**
ER1'	2.73%	5.21%	7.10%	11.29%	425	317
ER1'(UNITE)	**+0.00%**	**+5.72%**	**+7.11%**	**+10.63%**	**127**	**115**
GP19	2.73%	6.56%	12.57%	13.11%	417	278
GP19(UNITE)	**+0.55%**	**+6.98%**	**+6.18%**	**+10.39%**	**126**	**121**
Ochiai	2.73%	6.56%	13.70%	19.13%	397	227
Ochiai(UNITE)	**+0.55%**	**+9.29%**	**+4.01%**	**+7.1%**	**122**	**112**
MLP-FL	1.09%	4.32%	6.56%	9.29%	471	335
MLP-FL(UNITE)	**+1.10%**	**+7.16%**	**+8.19%**	**+11.48%**	**137**	**129**
CNN-FL	2.73%	6.01%	11.48%	17.53%	407	251
CNN-FL(UNITE)	**+0.55%**	**+10.93%**	**+6.55%**	**+3.78%**	**123**	**117**
BiLSTM-FL	1.09%	3.28%	6.15%	8.74%	493	354
BiLSTM-FL(UNITE)	**+1.10%**	**+7.65%**	**+8.06%**	**+14.18%**	**133**	**124**
ProFL	3.54%	7.64%	16.75%	25.95%	365	227
ProFL(UNITE)	**+0.78%**	**+9.85%**	**+6.75%**	**+4.1%**	**122**	**107**
DeepRL4FL	7.10%	15.41%	22.95%	26.23%	323	219
DeepRL4FL(UNITE)	**+0.00%**	**+4.81%**	**+3.83%**	**+6.56%**	**103**	**96**

Top-N Accuracy, MAR and MFR Parnin and Orso [38] conducted a user study of evaluating the usefulness of fault localization techniques in assisting developers, and recommended using the rank of the faulty statement to evaluate

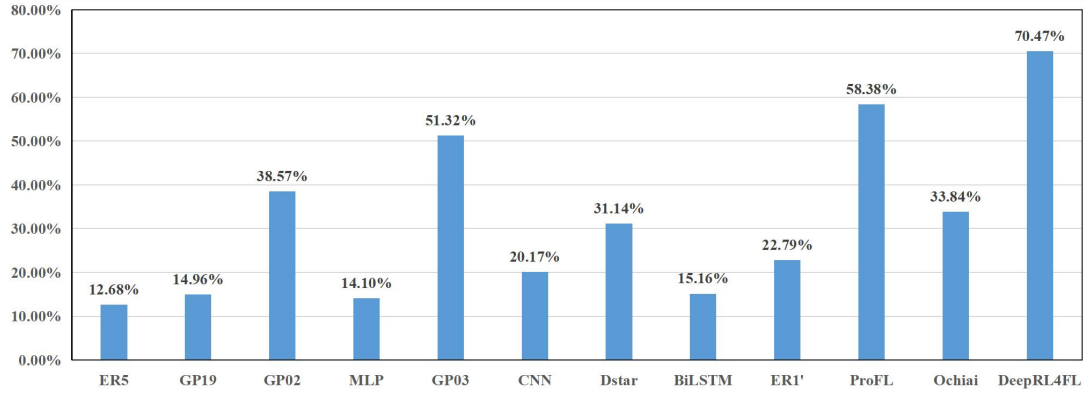

Fig. 6. *RImp* comparison of 12 FL baselines using UNITE vs without using UNITE.

fault localization effectiveness. Since then, *Top-N*, *MAR* and *MFR* are widely used in fault localization. Afterwards many comprehensive user studies (*e.g.,* [17, 58]) show that it is useful to help developers in debugging by using these metrics. Thus, our experiments use *Top-N*, *MAR*, and *MFR* to compare the 12 baselines between using UNITE and using the original representation. Table 3 presents their distribution among 12 fault localization approaches using original trace representation and UNITE representation, respectively. As shown in Table 3, UNITE achieves promising best localization effectiveness in all 12 scenarios in comparison to the baselines without using UNITE. Take one FL technique ER5 as an example. UNITE shows an increase of 1.64%, 9.84%, 8.20% and 9.91% improvement over ER5 for the Top-1, Top-3, Top-5 and Top-10 metrics respectively. The *MAR* and *MFR* are 134 and 125 respectively, achieving (421-134)/421=68.17% and (263-125)/263=52.47% relative improvement over ER5 respectively.

RImp distribution For a detailed improvement, we adopt *RImp* to evaluate UNITE. Fig. 6 shows the *RImp* distribution of UNITE: the *RImp* on the 12 FL baselines without using UNITE. As shown in Fig. 6, the *RImp* score is less than 100% in all approaches, meaning that UNITE improves localization effectiveness of all the 13 FL baselines. The statements that need to be examined decrease ranging from 14.10% in MLP-FL to 70.47% in DeepRL4FL. It also means that UNITE, obtains a maximum saving of 85.90% (100%-14.10%=85.90%) in MLP-FL and the minimum saving is 29.53% (100%-70.47%=48.68%) in DeepRL4FL, which indicates that UNITE can save from 29.53% to 85.90% of the number of statements examined among the fault localization approaches. Based on the *RImp* scores, we can observe that there is a significant saving after using UNITE, showing that UNITE is effective to improve fault localization.

Statistical comparison To investigate whether the difference between the baselines using UNITE and without using UNITE is statistically significant, we adopt Wilcoxon-Signed-Rank Test [5], with a Bonferroni correction [1], which is a non-parametric statistical hypothesis test for testing the differences between pairs of measurements F(x) and G(y). The experiments performed 12 paired Wilcoxon-Signed-Rank tests by using the *ranks* [30] of the faulty statements as the pairs of measurements F(x) and G(y). Each test uses left-tailed p-value checking at the σ level of 0.05. Specifically, we use the list of the *ranks* of the faulty statements using UNITE in all faulty versions of all programs as the list of measurements of F(x), while the list of measurements of G(y) is the list of the *ranks* of the faulty statements without using UNITE in all faulty versions of all programs. If p<0.05, H_1 that the *ranks* of using UNITE significantly tends to be smaller than that of without using UNITE is accepted, meaning that UNITE has BETTER effectiveness than without

Table 4. Wilcoxon-Signed-Rank Test results of the six of 12 FL approaches using UNITE vs without using UNITE (part 1).

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion
ER5 (UNITE) vs ER5	gzip	0.008	0.997	5.00e-04	BETTER	GP02 (UNITE) vs GP02	gzip	0.027	0.899	0.018	BETTER
	libtiff	0.018	0.978	0.005	BETTER		libtiff	0.011	0.969	0.009	BETTER
	python	0.010	0.958	0.012	BETTER		python	0.018	0.963	0.003	BETTER
	space	4.38e-04	1.000	2.41e-04	BETTER		space	4.38e-04	1.000	2.41e-04	BETTER
	nanoxml_v1	0.017	0.929	0.010	BETTER		nanoxml_v1	0.045	0.707	0.048	BETTER
	nanoxml_v2	0.018	0.963	0.019	BETTER		nanoxml_v2	0.039	0.789	0.041	BETTER
	nanoxml_v3	0.008	0.989	0.003	BETTER		nanoxml_v3	0.46	0.705	0.039	BETTER
	nanoxml_v5	0.013	0.985	0.007	BETTER		nanoxml_v5	0.009	0.985	3.01e-04	BETTER
	spoon	0.008	0.997	5.0e-04	BETTER		spoon	0.011	0.995	6.43e-03	BETTER
	dubbo	0.008	0.997	0.003	BETTER		dubbo	0.014	0.950	0.009	BETTER
	jackson-databind	0.013	0.929	0.009	BETTER		jackson-databind	0.025	0.896	0.023	BETTER
	oak	0.018	0.963	0.009	BETTER		oak	0.011	0.969	0.009	BETTER
	debezium	0.011	0.969	0.007	BETTER		debezium	0.018	0.966	0.003	BETTER
	byte-buddy	0.012	0.989	0.003	BETTER		byte-buddy	0.011	0.969	0.009	BETTER
	AutomatedCar	0.017	0.977	0.008	BETTER		AutomatedCar	0.018	0.963	0.019	BETTER
	cash-count	0.013	0.981	0.009	BETTER		cash-count	0.046	0.705	0.039	BETTER
	total	4.20e-12	1.000	2.15e-12	BETTER		total	2.99e-09	1.000	1.53e-09	BETTER
GP03 (UNITE) vs GP03	gzip	0.012	0.899	0.013	BETTER	Dstar (UNITE) vs Dstar	gzip	0.012	0.899	0.013	BETTER
	libtiff	0.011	0.969	0.009	BETTER		libtiff	0.011	0.969	0.09	BETTER
	python	0.018	0.967	4.00e-04	BETTER		python	0.018	0.963	3.00e-04	BETTER
	space	4.38e-04	1.000	2.41e-04	BETTER		space	4.38e-04	1.000	2.41e-04	BETTER
	nanoxml_v1	0.045	0.707	0.046	BETTER		nanoxml_v1	0.045	0.707	0.046	BETTER
	nanoxml_v2	0.011	0.969	0.009	BETTER		nanoxml_v2	0.043	0.789	0.039	BETTER
	nanoxml_v3	0.047	0.663	0.042	BETTER		nanoxml_v3	0.047	0.663	0.042	BETTER
	nanoxml_v5	0.013	0.985	0.003	BETTER		nanoxml_v5	0.013	0.985	0.003	BETTER
	spoon	0.008	1.000	4.58e-04	BETTER		spoon	0.008	0.962	0.007	BETTER
	dubbo	0.008	0.970	0.005	BETTER		dubbo	0.045	0.728	0.041	BETTER
	jackson-databind	0.012	0.989	0.003	BETTER		jackson-databind	0.024	0.896	0.035	BETTER
	oak	0.011	0.961	0.009	BETTER		oak	0.015	0.909	0.010	BETTER
	debezium	0.018	0.963	0.003	BETTER		debezium	0.007	0.985	0.003	BETTER
	byte-buddy	0.011	0.969	0.014	BETTER		byte-buddy	0.045	0.707	0.048	BETTER
	AutomatedCar	0.011	0.969	0.009	BETTER		AutomatedCar	0.018	0.963	0.003	BETTER
	cash-count	0.046	0.705	0.039	BETTER		cash-count	0.047	0.663	0.042	BETTER
	total	2.32e-08	1.000	1.25e-08	BETTER		total	1.61e-07	1.000	8.19e-08	BETTER
ER1' (UNITE) vs ER1'	gzip	0.017	0.899	0.016	BETTER	GP19 (UNITE) vs GP19	gzip	0.008	0.978	0.007	BETTER
	libtiff	0.011	0.969	0.009	BETTER		libtiff	0.11	0.969	0.009	BETTER
	python	0.018	0.963	0.003	BETTER		python	0.018	0.963	0.003	BETTER
	space	4.38e-04	1.000	2.41e-04	BETTER		space	4.38e-04	1.000	2.41e-04	BETTER
	nanoxml_v1	0.045	0.707	0.047	BETTER		nanoxml_v1	0.011	0.966	0.011	BETTER
	nanoxml_v2	0.011	0.969	0.009	BETTER		nanoxml_v2	0.011	0.969	0.009	BETTER
	nanoxml_v3	0.045	0.663	0.042	BETTER		nanoxml_v3	0.012	0.953	0.009	BETTER
	nanoxml_v5	0.003	0.985	0.008	BETTER		nanoxml_v5	0.003	0.985	0.008	BETTER
	spoon	0.008	0.962	0.004	BETTER		spoon	0.008	0.997	0.005	BETTER
	dubbo	0.045	0.702	0.041	BETTER		dubbo	0.015	0.911	0.014	BETTER
	jackson-databind	0.25	0.896	0.042	BETTER		jackson-databind	0.013	0.929	0.010	BETTER
	oak	0.015	0.909	0.009	BETTER		oak	0.019	0.789	0.039	BETTER
	debezium	0.009	0.985	0.003	BETTER		debezium	0.012	0.953	0.010	BETTER
	byte-buddy	0.010	0.969	0.009	BETTER		byte-buddy	0.010	0.969	0.009	BETTER
	AutomatedCar	0.007	0.989	0.003	BETTER		AutomatedCar	0.012	0.963	0.003	BETTER
	cash-count	0.025	0.896	0.043	BETTER		cash-count	0.046	0.705	0.039	BETTER
	total	3.25e-08	1.000	1.66e-08	BETTER		total	4.79e-11	1.000	2.45e-11	BETTER

using UNITE; otherwise, H_0 that *ranks* of using UNITE does not significantly tend to be smaller than that of without using UNITE is accepted, meaning that using UNITE does not perform better than without using UNITE.

Table 4 and Table 5 show the Wilcoxon-Signed-Rank Test results on this relationship, where the cells show the p values of Wilcoxon-Signed-Rank Tests. The results show that the *ranks* of the faulty statements of all the 12 FL approaches using UNITE are significantly smaller than those of all the 12 baselines using original trace representation in all programs, yielding BETTER results in all cases.

To further assess the difference quantitatively, we leverage the nonparametric Vargha-Delaney A-test, which is recommended in [3], to evaluate the magnitude of the difference by measuring effect size (scientific significance). For A-test, the bigger deviation of A-statistic is from the value of 0.5, the greater difference is between the two studied groups. Vargha and Delaney [50] suggest that A-test of greater than 0.64 (or less than 0.36) is indicative of "medium" effect size, and of greater than 0.71 (or less than 0.29) can be indicative of a promising "large" effect size.

Table 5. Wilcoxon-Signed-Rank Test results of the other six of 12 FL approaches using UNITE vs without using UNITE (part 2).

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion
Ochiai (UNITE) vs Ochiai	gzip	0.008	0.978	0.007	BETTER	MLP (UNITE) vs MLP	gzip	0.008	0.978	0.005	BETTER
	libtiff	0.011	0.969	0.004	BETTER		libtiff	0.011	0.966	0.009	BETTER
	python	0.002	0.993	0.002	BETTER		python	0.012	0.963	0.009	BETTER
	space	4.38e-04	1.000	2.41e-04	BETTER		space	4.38e-04	1.000	2.41e-04	BETTER
	nanoxml_v1	0.042	0.707	0.047	BETTER		nanoxml_v1	0.009	0.971	0.009	BETTER
	nanoxml_v2	0.033	0.789	0.039	BETTER		nanoxml_v2	0.013	0.961	0.011	BETTER
	nanoxml_v3	0.034	0.853	0.020	BETTER		nanoxml_v3	0.012	0.989	0.003	BETTER
	nanoxml_v5	0.005	0.985	0.003	BETTER		nanoxml_v5	0.005	0.985	0.003	BETTER
	spoon	0.006	0.971	0.004	BETTER		spoon	0.015	0.909	0.011	BETTER
	dubbo	0.045	0.819	0.029	BETTER		dubbo	0.017	0.943	0.019	BETTER
	jackson-databind	0.013	0.949	0.010	BETTER		jackson-databind	0.008	0.970	0.005	BETTER
	oak	0.019	0.789	0.039	BETTER		oak	0.018	0.963	0.009	BETTER
	debezium	0.010	0.969	0.009	BETTER		debezium	0.012	0.965	0.003	BETTER
	byte-buddy	0.010	0.969	0.009	BETTER		byte-buddy	0.005	0.989	0.002	BETTER
	AutomatedCar	0.010	0.923	0.013	BETTER		AutomatedCar	0.002	0.991	0.002	BETTER
	cash-count	0.024	0.896	0.025	BETTER		cash-count	0.006	0.985	0.003	BETTER
	total	3.90e-09	1.000	1.99e-09	BETTER		total	3.55e-10	1.000	1.83e-10	BETTER
CNN (UNITE) vs CNN	gzip	0.008	0.978	0.005	BETTER	BiLSTM (UNITE) vs BiLSTM	gzip	0.007	0.978	0.005	BETTER
	libtiff	0.011	0.969	0.004	BETTER		libtiff	0.011	0.969	0.004	BETTER
	python	0.012	0.952	0.003	BETTER		python	0.002	0.998	3.25e-04	BETTER
	space	4.36e-04	1.000	2.41e-04	BETTER		space	4.38e-04	1.000	2.41e-04	BETTER
	nanoxml_v1	0.011	0.957	0.006	BETTER		nanoxml_v1	0.011	0.957	0.006	BETTER
	nanoxml_v2	0.011	0.956	0.010	BETTER		nanoxml_v2	0.011	0.956	0.010	BETTER
	nanoxml_v3	0.011	0.953	0.007	BETTER		nanoxml_v3	0.011	0.953	0.007	BETTER
	nanoxml_v5	0.006	0.985	0.003	BETTER		nanoxml_v5	0.006	0.985	0.003	BETTER
	spoon	0.010	0.969	0.004	BETTER		spoon	0.009	0.909	0.011	BETTER
	dubbo	0.008	0.951	0.019	BETTER		dubbo	0.008	0.972	0.003	BETTER
	jackson-databind	0.036	0.791	0.029	BETTER		jackson-databind	0.004	0.985	0.006	BETTER
	oak	0.008	0.942	0.019	BETTER		oak	0.008	0.981	0.003	BETTER
	debezium	0.004	0.985	0.003	BETTER		debezium	0.011	0.964	0.003	BETTER
	byte-buddy	0.008	0.970	0.005	BETTER		byte-buddy	0.006	0.978	0.004	BETTER
	AutomatedCar	0.006	0.971	0.005	BETTER		AutomatedCar	0.018	0.913	0.003	BETTER
	cash-count	0.004	0.985	0.003	BETTER		cash-count	0.006	0.978	0.005	BETTER
	total	9.50e-10	1.000	4.88e-10	BETTER		total	4.19e-10	1.000	2.15e-10	BETTER
ProFL (UNITE) vs ProFL	gzip	0.008	0.978	0.007	BETTER	DeepRL4FL (UNITE) vs DeepRL4FL	gzip	0.009	0.969	0.011	BETTER
	libtiff	0.009	0.969	0.011	BETTER		libtiff	0.003	0.989	0.004	BETTER
	python	0.011	0.963	0.009	BETTER		python	0.035	0.859	0.021	BETTER
	space	4.36e-04	1.000	2.41e-04	BETTER		space	4.36e-04	1.000	2.41e-04	BETTER
	nanoxml_v1	0.009	0.969	0.011	BETTER		nanoxml_v1	0.35	0.708	0.043	BETTER
	nanoxml_v2	0.009	0.978	0.004	BETTER		nanoxml_v2	0.007	0.973	0.003	BETTER
	nanoxml_v3	0.011	0.953	0.007	BETTER		nanoxml_v3	0.025	0.814	0.015	BETTER
	nanoxml_v5	0.004	0.985	0.003	BETTER		nanoxml_v5	0.015	0.896	0.027	BETTER
	spoon	0.009	0.987	0.003	BETTER		spoon	0.035	0.789	0.039	BETTER
	dubbo	0.035	0.789	0.039	BETTER		dubbo	0.004	0.989	0.002	BETTER
	jackson-databind	0.015	0.896	0.024	BETTER		jackson-databind	0.005	0.969	0.010	BETTER
	oak	0.005	0.969	0.010	BETTER		oak	0.008	0.981	0.003	BETTER
	debezium	0.008	0.978	0.006	BETTER		debezium	0.008	0.972	0.009	BETTER
	byte-buddy	0.003	0.979	0.007	BETTER		byte-buddy	0.003	0.989	0.002	BETTER
	AutomatedCar	0.003	0.989	0.002	BETTER		AutomatedCar	0.042	0.705	0.043	BETTER
	cash-count	0.015	0.896	0.015	BETTER		cash-count	0.012	0.911	0.004	BETTER
	total	1.31e-07	1.000	7.35e-08	BETTER		libtiff	1.65e-07	1.000	6.36e-08	BETTER

Table 6. A-Test results of 12 FL approaches using UNITE vs without using UNITE.

Comparison	A-Test	Comparison	A-Test	Comparison	A-Test	Comparison	A-Test
ER5(UNITE) vs ER5	0.86	GP02(UNITE) vs GP02	0.87	GP03(UNITE) vs GP03	0.91	Dstar(UNITE) vs Dstar	0.85
ER1'(UNITE) vs ER1'	0.88	GP19(UNITE) vs GP19	0.91	Ochiai(UNITE) vs Ochiai	0.88	MLP(UNITE) vs MLP	0.93
CNN(UNITE) vs CNN	0.89	BiLSTM(UNITE) vs BiLSTM	0.96	ProFL(UNITE) vs ProFL	0.83	DeepRL4FL(UNITE) vs DeepRL4FL	0.81

Table 6 shows the A-Test results of 12 FL approaches using UNITE vs without using UNITE. We could observe that UNITE arrives at the promising "large" effect size, thus showing better performance. Therefore, it is statistically significant that UNITE outperforms FL without using UNITE.

Summary for RQ1 *In RQ1, we explore the effectiveness of UNITE over original 12 FL baselines. We can safely conclude that the 12 techniques with UNITE significantly outperform the original ones, showing that incorporating influential global and local contexts guided trace representation into FL is potential to improve FL effectiveness.*

Table 7. Statistical results of the 12 FL approaches using each part of UNITE vs using original representation.

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test	Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test
UNITE (Influence) vs original representation	ER5	0.010	0.978	0.009	BETTER	0.83	UNITE (GLContexts) vs original representation	ER5	0.018	0.968	0.009	BETTER	0.73
	GP02	0.012	0.785	0.018	BETTER	0.78		GP02	0.014	0.917	0.003	BETTER	0.74
	GP03	0.026	0.865	0.004	BETTER	0.67		GP03	0.015	0.935	0.003	BETTER	0.74
	Dstar	0.027	0.893	0.031	BETTER	0.65		Dstar	0.034	0.746	0.026	BETTER	0.63
	ER1'	0.009	0.903	0.004	BETTER	0.81		ER1'	0.034	0.743	0.029	BETTER	0.63
	GP19	0.015	0.899	0.028	BETTER	0.64		GP19	0.031	0.824	0.046	BETTER	0.63
	Ochiai	0.015	0.912	0.005	BETTER	0.64		Ochiai	0.012	0.876	0.009	BETTER	0.76
	MLP-FL	0.011	0.969	0.009	BETTER	0.87		MLP-FL	0.043	0.785	0.030	BETTER	0.65
	CNN-FL	0.013	0.902	0.018	BETTER	0.62		CNN-FL	0.028	0.897	0.029	BETTER	0.76
	BiLSTM-FL	0.010	0.974	0.002	BETTER	0.81		BiLSTM-FL	0.016	0.894	0.008	BETTER	0.78
	ProFL	0.017	0.929	0.014	BETTER	0.67		ProFL	0.038	0.770	0.037	BETTER	0.62
	DeepRL4FL	0.026	0.893	0.025	BETTER	0.63		DeepRL4FL	0.045	0.812	0.039	BETTER	0.61

Table 8. Statistical results of the 12 FL approaches using UNITE vs using each part of UNITE.

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test	Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test
UNITE vs UNITE (Influence)	ER5	0.013	0.909	0.004	BETTER	0.74	UNITE vs UNITE (GLContexts)	ER5	0.016	0.913	0.005	BETTER	0.71
	GP02	0.018	0.903	0.009	BETTER	0.82		GP02	0.011	0.869	0.041	BETTER	0.73
	GP03	0.029	0.859	0.021	BETTER	0.71		GP03	0.015	0.835	0.038	BETTER	0.74
	Dstar	0.023	0.893	0.020	BETTER	0.73		Dstar	0.014	0.846	0.036	BETTER	0.71
	ER1'	0.039	0.824	0.043	BETTER	0.71		ER1'	0.014	0.885	0.039	BETTER	0.72
	GP19	0.016	0.906	0.009	BETTER	0.81		GP19	0.011	0.895	0.009	BETTER	0.81
	Ochiai	0.015	0.903	0.005	BETTER	0.78		Ochiai	0.012	0.911	0.016	BETTER	0.73
	MLP-FL	0.010	0.969	0.008	BETTER	0.82		MLP-FL	0.013	0.907	0.012	BETTER	0.71
	CNN-FL	0.031	0.864	0.046	BETTER	0.72		CNN-FL	0.018	0.847	0.036	BETTER	0.71
	BiLSTM-FL	0.017	0.915	0.014	BETTER	0.73		BiLSTM-FL	0.013	0.902	0.014	BETTER	0.78
	ProFL	0.013	0.919	0.009	BETTER	0.72		ProFL	0.019	0.886	0.046	BETTER	0.72
	DeepRL4FL	0.017	0.899	0.024	BETTER	0.72		DeepRL4FL	0.017	0.878	0.045	BETTER	0.73

4.3.2 *RQ2.* **Does each part of UNITE contribute to FL effectiveness?**

UNITE has two major parts: combining global and local contexts into representation and incorporating influential semantics into representation. It is desirable to see whether each part of UNITE contributes to FL effectiveness. Therefore, We implement UNITE with each part as UNITE(GLContexts) and UNITE(Influence), respectively. There are two cases: (1) we compare UNITE(GLContexts) and UNITE(Influence) with the original trace representation to check whether each part improves the original one; (2) we compare UNITE with each part (*i.e.,* UNITE(GLContexts) and UNITE(Influence)) to check whether UNITE successfully combines two parts to achieve better effectiveness than each part. We use the *ranks* of the faulty statements as measurements, and conduct Wilcoxon-Signed-Rank Test with a Bonferroni correction at the σ level of 0.05 for each comparison of the above two cases. Furthermore, for each comparison of the above two cases, we adopt the nonparametric Vargha-Delaney A-test to evaluate the magnitude of their difference by measuring effect size.

Table 7 and Table 8 show the statistical results of each one of the above two cases, respectively. As shown in Table 7, the *ranks* of the faulty statements of all the 12 FL baselines using each part of UNITE (*i.e.,* UNITE(GLContexts) and UNITE(Influence)) are significantly smaller than those of all the original FL approaches, yielding BETTER results in all scenarios. Furthermore, each part of UNITE (*i.e.,* UNITE(GLContexts) and UNITE(Influence)) acquire "medium" and "large" effect sizes over those of all original FL approaches. Similarly, as show in Table 8, UNITE significantly outperforms its each part (*i.e.,* UNITE(GLContexts) and UNITE(Influence)), yielding BETTER results and "large" effect sizes in all scenarios.

Summary for RQ2 *In RQ2, we explore the contribution of each part of UNITE to FL effectiveness. Based on the above results, we can conclude that (1) each part of UNITE (*i.e., *UNITE(GLContexts) and UNITE(Influence)) significantly contributes to FL effectiveness; (2) UNITE successfully combines the contributions of UNITE(GLContexts) and UNITE(Influence), significantly outperforming each separated part.*

4.3.3 *RQ3.* Why is UNITE better than original state-of-the-art FL baselines?

The experimental results show that UNITE outperforms the original trace representation. It is natural to seek why is UNITE better than original trace representation. Let us use the definitions (*e.g.*, x_{ij} and $GL_{influence}(x_{ij})$) in Section 3. For a statement s_j, we first define the following four formulas:

$$\begin{aligned} fWeights_{origin}(s_j) &= \sum_{i \in \{i|e_i=1\}} x_{ij} \\ fWeights_{UNITE}(s_j) &= \sum_{i \in \{i|e_i=1\}} GL_{influence}(x_{ij}) \\ pWeights_{origin}(s_j) &= \sum_{i \in \{i|e_i=0\}} x_{ij} \\ pWeights_{UNITE}(s_j) &= \sum_{i \in \{i|e_i=0\}} GL_{influence}(x_{ij}) \end{aligned} \tag{10}$$

$fWeights_{origin}(s_j)$ and $fWeights_{UNITE}(s_j)$ denote the cumulative weights of the statement s_j acquired in all failing test cases by using original representation and UNITE, respectively. Similarly, $pWeights_{origin}(s_j)$ and $pWeights_{UNITE}(s_j)$ represent the cumulative weights of the statement s_j acquired in all passing test cases by using original representation and UNITE, respectively. For a statement, a high $fWeights$ means that it is strongly related to failing test cases whereas a high $pWeights$ represents it is strongly related to passing test cases[10]. Thus, it desirable to design a trace representation that will always assign the faulty statements with a high $fWeights$ and a low $pWeights$. This may be the reason why UNITE outperforms the original trace representation. In other words, the values of the *ranks* and *exam* of the faulty statements in descending order of $fWeights_{UNITE}$ should be smaller than those in descending order of $fWeights_{origin}$, and the values of the ranks of the faulty statements in descending order of $pWeights_{UNITE}$ should be higher than those in descending order of $pWeights_{origin}$.

To verify the above analysis, based on the four formulas in Eq. (10), we calculate the $fWeights$ and $pWeights$ of each statement in all faulty versions of a program using the original representation and UNITE, respectively. We conduct two paired Wilcoxon-Signed-Rank tests with a Bonferroni correction by using the pairs of measurements F(x) and G(y), and each test uses left-tailed p-value checking at the σ level of 0.05. One test adopts the *ranks* and *exam* of the faulty statements using $fWeights_{UNITE}$ in all faulty versions of a program as the list of measurements of F(x), while the list of measurements of G(y) is the list of the *ranks* and *exam* of the faulty statements using $fWeights_{origin}$ in all faulty versions of the program. The other test utilizes the ranks of the faulty statements using $pWeights_{orgin}$ in all faulty versions of a program as the list of measurements of F(x), while the list of measurements of G(y) is the list of the *ranks* and *exam* of the faulty statements using $pWeights_{UNITE}$ in all faulty versions of the program. For each of the above comparison, we further adopt the nonparametric Vargha-Delaney A-test to evaluate the magnitude of their difference by measuring effect size.

[10]This analysis excludes those statement whose $fWeights$ and $pWeights$ are both 0 because they have nothing with failing and passing test cases and will be first excluded by FL techniques.

Table 9. Statistical results of the comparison between UNITE and the original representation using $fWeights$ and $pWeights$.

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test
$fWeights_{UNITE}$ vs $fWeights_{orgin}$	gzip	0.014	0.939	0.001	BETTER	0.81
	libtiff	0.016	0.921	0.007	BETTER	0.78
	python	0.007	0.965	0.001	BETTER	0.82
	space	0.012	0.914	0.003	BETTER	0.76
	nanoxml_v1	0.011	0.951	0.007	BETTER	0.77
	nanoxml_v2	0.012	0.916	0.008	BETTER	0.76
	nanoxml_v3	0.011	0.903	0.009	BETTER	0.78
	nanoxml_v5	0.010	0.925	0.008	BETTER	0.76
	spoon	0.004	0.985	0.003	BETTER	0.85
	dubbo	0.011	0.924	0.008	BETTER	0.78
	jackson-databind	0.018	0.892	0.026	BETTER	0.76
	oak	0.017	0.914	0.002	BETTER	0.74
	debezium	0.008	0.970	0.003	BETTER	0.82
	byte-buddy	0.012	0.893	0.022	BETTER	0.77
	AutomatedCar	0.016	0.907	0.012	BETTER	0.76
	cash-count	0.017	0.879	0.029	BETTER	0.76
Comparison		**2-tailed**	**1-tailed(right)**	**1-tailed(left)**	**Conclusion**	**A-Test**
$pWeights_{UNITE}$ vs $pWeights_{origin}$	gzip	0.012	0.953	0.007	BETTER	0.74
	libtiff	0.033	0.865	0.037	BETTER	0.71
	python	0.018	0.906	0.009	BETTER	0.76
	space	0.029	0.899	0.038	BETTER	0.72
	nanoxml_v1	0.025	0.883	0.046	BETTER	0.72
	nanoxml_v2	0.025	0.894	0.041	BETTER	0.74
	nanoxml_v3	0.023	0.917	0.024	BETTER	0.74
	nanoxml_v5	0.020	0.953	0.006	BETTER	0.76
	spoon	0.012	0.957	0.007	BETTER	0.77
	dubbo	0.033	0.889	0.038	BETTER	0.72
	jackson-databind	0.036	0.846	0.037	BETTER	0.71
	oak	0.016	0.908	0.007	BETTER	0.76
	debezium	0.018	0.917	0.014	BETTER	0.75
	byte-buddy	0.037	0.802	0.043	BETTER	0.71
	AutomatedCar	0.028	0.835	0.038	BETTER	0.72
	cash-count	0.034	0.849	0.045	BETTER	0.71

Table 9 shows the statistical results of the comparison between UNITE and original trace representation using failing and passing cumulative weights, respectively. As shown in Table 9, the values of the *ranks* of the faulty statements using $fWeights_{UNITE}$ and $pWeights_{orgin}$ are significantly smaller than $fWeights_{origin}$ and $pWeights_{UNITE}$, respectively, yielding BETTER results and "large" effect sizes in all programs.

Summary for RQ3 *In RQ3, we explore the reason of why UNITE performs better than original FL techniques. The results show that the reason of UNITE outperforms the original trace representation may lie in that UNITE will always assign the faulty statements with a high $fWeights$ and a low $pWeights$.*

5 DISCUSSION

5.1 Benchmark Over-fitting Effect on UNITE

Does benchmark over-fitting effect impact UNITE? Recent work [10] shows that the widely-used benchmark Defects4J (*i.e., chart*, *math*, *lang* and *time* in the Table 2) is over-fitting for SFL including the seven state-of-the-art FL baselines (*i.e.,* ER5, GP02, GP03, Dstar, ER1', GP19 and Ochiai) used by our experiments. In other words, SFL shows inconsistencies between the benchmark Defects4J and other benchmarks in terms of FL effectiveness. For example, 34.8% and 47.8% of bugs in Defects4J are localized at top 10 using Ochiai and Dstar while only a few bugs in other benchmarks can be localized even in top 100 [10, 72]. It is interesting to see whether UNITE still effectively works under the effect of benchmark over-fitting.

Table 10. The statistical results of the 12 FL approaches on Defects4J using UNITE vs without using UNITE.

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test
UNITE vs original representation	ER5	0.017	0.914	0.017	BETTER	0.78
	GP02	0.019	0.895	0.038	BETTER	0.76
	GP03	0.022	0.879	0.014	BETTER	0.74
	Dstar	0.010	0.913	0.013	BETTER	0.75
	ER1'	0.013	0.902	0.012	BETTER	0.76
	GP19	0.012	0.914	0.018	BETTER	0.77
	Ochiai	0.013	0.927	0.010	BETTER	0.76
	MLP-FL	0.016	0.925	0.014	BETTER	0.75
	CNN-FL	0.012	0.894	0.026	BETTER	0.77
	BiLSTM-FL	0.009	0.966	0.008	BETTER	0.82
	ProFL	0.025	0.843	0.046	BETTER	0.74
	DeepRL4FL	0.027	0.834	0.047	BETTER	0.72

Table 11. Statistical results of the comparison between UNITE and the original representation using $fWeights$ and $pWeights$ on Defects4J.

Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test	Comparison		2-tailed	1-tailed(right)	1-tailed(left)	Conclusion	A-Test
$fWeights_{UNITE}$ vs $fWeights_{origin}$	chart	0.012	0.906	0.009	BETTER	0.79	$pWeights_{UNITE}$ vs $pWeights_{origin}$	chart	0.024	0.879	0.034	BETTER	0.74
	math	0.010	0.916	0.008	BETTER	0.79		math	0.022	0.898	0.014	BETTER	0.76
	lang	0.008	0.946	0.003	BETTER	0.78		lang	0.011	0.967	0.009	BETTER	0.81
	time	0.011	0.906	0.009	BETTER	0.78		time	0.008	0.963	0.006	BETTER	0.78

We apply UNITE to the 12 FL techniques on Defects4J, and compare their FL effectiveness. Specifically, we perform 12 paired Wilcoxon-Signed-Rank tests by using the *ranks* and *exam* of the faulty statements as the pairs of measurements F(x) (*i.e.,* UNITE) and G(y) (*i.e.,* each of 12 original FL baselines). Each test uses left-tailed p-value checking at the σ level of 0.05.

Table 10 shows the statistical results on this relationship. As shown in Table 10, the p-values are all less than 0.05 and the A-test values are all greater than 0.71. It means that the *ranks* of the faulty statements of all the 12 FL approaches using UNITE are significantly smaller than those of all the original FL approaches on Defects4J, yielding BETTER results and "large" effect sizes in all scenarios. Thus, UNITE can still effectively work under the effect of benchmark over-fitting.

Does the reason of a high fWeights and a low pWeights still work for UNITE under the benchmark over-fitting effect? In RQ3, the results show that the reason of UNITE outperforms the original trace representation may lie in that UNITE will always assign the faulty statements with a high $fWeights$ and a low $pWeights$. This reason may still work for explaining that the effect of benchmark over-fitting does not impact UNITE. Thus, we also conduct two paired Wilcoxon-Signed-Rank tests with a Bonferroni correction on Defects4J by using left-tailed p-value checking at the σ level of 0.05. One test adopts the ranks of the faulty statements using $fWeights_{UNITE}$ in all faulty versions of Defects4J as the list of measurements of F(x), while the list of measurements of G(y) is the list of the *ranks* of the faulty statements using $fWeights_{origin}$ in all faulty versions of Defects4J. The other test utilizes the ranks of the faulty statements using $pWeights_{orgin}$ in all faulty versions of Defects4J as the list of measurements of F(x), while the list of measurements of G(y) is the list of the *ranks* of the faulty statements using $pWeights_{UNITE}$ in all faulty versions of Defects4J.

Table 11 shows the statistical results of the comparison between UNITE and original trace representation using $fWeights$ and $pWeights$ on Defects4J, respectively. As shown in Table 11, the p-values are all less than 0.05 and the A-test values are all greater than 0.71, yielding BETTER results and "large" effect sizes in all programs of Defects4J. Thus, under the effect of benchmark over-fitting, UNITE will still always assign the faulty statements with a high $fWeights$ and a low $pWeights$.

Table 12. Average time cost of using UNITE and without using UNITE.

Comparison	ER5(UNITE)/ER5	GP02(UNITE)/GP02	GP03(UNITE)/GP03	Dstar(UNITE)/Dstar
Time Cost	35.7s/4.7s	31.8s/4.2s	41.2s/5.3s	32.3s/4.4s
Comparison	ER1'(UNITE)/ER1'	GP19(UNITE)/GP19	Ochiai(UNITE)/Ochiai	MLP-FL(UNITE)/MLP-FL
Time Cost	44.7s/5.9s	45.3s/6.2s	36.1s/4.9s	3.5h/2.1h
Comparison	CNN-FL(UNITE)/CNN-FL	BiLSTM-FL(UNITE)/BiLSTM-FL	ProFL(UNITE)/PRoFL	DeepRL4FL(UNITE)/DeepRL4FL
Time Cost	5.9h/4.1h	18.7h/11.3h	3.6h/2.3h	6.2h/4.6h

5.2 Efficiency of UNITE

Due to the use of both global and local contexts with influential semantics, it is necessary to evaluate the efficiency of UNITE. Table 12 shows the average time cost of 12 baselines using and without using UNITE, where s and h denote seconds and hours respectively. As shown in Table 12, for the seven baselines (*i.e.,* ER5, GP02, GP03, Dstar, ER1', GP19 and Ochiai), even if the time cost changes from several seconds into dozens of seconds after using UNITE, the time cost is still low. For the other baselines (*i.e.,* MLP-FL, CNN-FL, BiLSTM-FL, ProFL, DeepFL4FL), the time costs of using UNITE and without using UNITE are within the same order of magnitude. Thus, the time cost of UNITE is acceptable in comparison to the original baselines.

5.3 Application of UNITE in Automated Program Repair

Automated program repair (APR) [19] is a concrete software engineering task by automatically repairing programs. APR usually consists of three phases: fault localization, patch generation and patch validation. Being the first step, fault localization provides a suspicious rank list of statements for APR. Specifically, the APR techniques generate patches in the suspicious rank list from top to down and many APR techniques [12, 27, 62] set clear time limitation. It means that, after using UNITE, the improvement of *Top-N* and *MFR* metrics could help the APR techniques, since the APR techniques relay on the suspicious rank list and have limited time for each bug during the repair. Thus, we adopt the concrete software engineering task(*i.e.,* APR) to illustrate meaningful improvement of our approach.

We use two typical APR techniques (*i.e.,* Nopol [62] and Tbar [27]) and apply UNITE to their fault localization modules (*i.e.,* Ochiai [35]). We adopt Defects4J, widely used by the existing APR studies including Nopol [62] and Tbar [27], to conduct the comparison. We further exclude those faulty versions which the slicing tools cannot slice, and apply Nopol and Tbar to these faulty versions, where Nopol generated plausible patches for the programs of chart, lang and math and Tbar produced plausible patches for the programs of chart, lang, math and time. Thus, we perform 100 repeated repairs for each of those faulty versions which are finally fixed by Nopol [62] or Tbar [27].

To evaluate the effect of UNITE on APR efficiency, we adopt two widely used metrics (*i.e., repair time* and *NPC*) [27, 62]. We show different parts of *repair time* in seconds: fault localization time (*i.e.,* the time cost of fault localization), patch acquisition time (*i.e.,* the time cost of patch generation and validation), total time (*i.e.,* the time cost of the whole APR process including fault localization time and patch acquisition time). *NPC* denotes the number of patch candidates generated by an APR technique until the first plausible patch is found. Table 13 shows the efficiency distribution of APR techniques with and without using UNITE. As shown in Table 13, for *repair time*, although our approach increases fault localization time, the patch acquisition time decreases and the total time decreases except for two programs using Nopol; for NPC, our approach reduces the NPC in Tbar and keeps the same NPC in Nopol. These results show that UNITE can improve the APR efficiency.

Table 13. Efficiency distribution of repair time and NPC among the original APR techniques and the ones using UNITE.

Comparison		Fault Localization Time (s)	Patch Acquisition Time (s)	Total Time (s)	NPC
Nopol	chart	13.15	5.69	19.84	1
	lang	15.38	29.30	44.68	1
	math	69.84	293.45	363.29	1
Nopol(UNITE)	chart	47.37	3.25	50.62	1
	lang	52.64	18.67	71.31	1
	math	121.43	238.91	360.34	1
TBar	chart	15.79	783.75	799.54	587.25
	lang	16.54	848.49	865.03	714.75
	math	86.81	798.09	884.90	79.60
	time	12.08	7244.91	7256.99	6812.63
Tbar(UNITE)	chart	56.23	542.61	598.84	327.39
	lang	57.19	585.83	643.02	485.92
	math	142.25	505.78	648.03	61.37
	time	47.85	4473.38	4521.23	2341.25

Table 14. Effectiveness distribution of plausibly fixed bugs among the original APR techniques and the ones using UNITE.

Comparison		Fixed Bugs
Nopol	chart	5,9,13,17
	lang	44,51,58
	math	40
Nopol(UNITE)	chart	5,9,13,17
	lang	44,51,58
	math	40,**50**
TBar	chart	1,4,7,8,9,11,12,13,14,15,19,20,24,25
	lang	7,10,22,33,39,43,44,45,47,51,58,59,63
	math	2,3,4,5,6,8,11,15,22,28,30,32,33,34,35,49,50,57,58,59,60,62,63,65,70,73,75,77,79,80,82,85,89,95,96,98
	time	7,11,17
Tbar(UNITE)	chart	1,4,7,8,9,11,12,13,14,15,19,20,24,25,**26**
	lang	7,10,22,33,39,43,44,45,47,51,58,59,63, **13**, **18**, **27**
	math	2,3,4,5,6,8,11,15,22,28,30,32,33,34,35,49,50,57,58,59,60,62,63,65,70,73,75,77,79,80,82,85,89,95,96,98,**52**,**88**,**94**
	time	7,11,17,**2**,**19**

To evaluate the effect of UNITE on the APR effectiveness, we adopt the widely used metric, *i.e.,* the number of plausibly fixed bugs generated by an APR technique [27, 62]. Table 14 shows the specific fixed bugs of the original APR techniques and the ones using UNITE. As shown in Table 14, after applying UNITE, for Nopol, it has plausibly fixed one more bug (*i.e.,* math_50); for Tbar, it has plausibly fixed nine more bugs (*i.e.,* chart_26, lang_13, lang_18, lang_27, math_52, math_88, math_94, time_2 and time_19). Thus, UNITE can improves the APR effectiveness.

5.4 An Example of Qualitative Analysis for UNITE

To show whether the difference is meaningful after applying UNITE, we demonstrate a qualitative example to show the detailed information of 12 FL approaches locating the faults. Specifically, we use the faulty version two of the program nanoxml_v2 whose faulty statement is the line 309 as the qualitative example, showing the faulty program with call relationship and the locations where the 12 FL approaches locate the faults.

Table 15 summarized the detailed results of 12 FL approaches with and without UNITE, where the column 'Ranking List' is the ranking list of the statements in descending order of suspiciousness until finding the faulty statement and the column 'Rank' denotes the rank of faulty statement in the ranking list. As shown in Table 15, after applying UNITE, the length of the ranking list decreases and the rank of the faulty statement increases, showing UNITE is more effective.

Table 15. Detailed FL results of the qualitative example.

Comparison	Ranking list	Rank
ER5	52 58 64 96 97 98 99 100 101 125 126 147 148 191 192 218 220 221 239 240 243 245 246 253 256 263 277 279 285 286 289 290 292 331 333 344 345 348 349 **390**	40
ER5(UNITE)	344 **390**	2
GP02	348 349 344 345 **390**	5
GP02(UNITE)	344 **390**	2
GP03	348 349 344 345 **390**	5
GP03(UNITE)	344 **390**	2
Dstar	348 349 344 345 **390**	5
Dstar(UNITE)	344 **390**	2
ER1'	348 349 344 345 **390**	5
ER1'(UNITE)	344 **390**	2
GP19	52 58 64 96 97 98 99 100 101 125 126 147 148 191 192 239 240 243 277 279 453 455 456 457 459 464 467 473 475 292 484 485 489 491 505 507 256 590 592 594 348 349 245 246 479 480 481 600 602 605 606 289 290 476 522 344 345 **390**	58
GP19(UNITE)	344 **390**	2
Ochiai	348 349 344 345 **390**	5
Ochiai(UNITE)	344 **390**	2
MLP-FL	125 126 191 218 220 221 239 240 453 455 456 457 459 243 344 345 **390**	17
MLP-FL(UNITE)	218 220 239 240 243 344 **390**	7
CNN-FL	239 240 243 277 279 344 345 **390**	8
CNN-FL(UNITE)	243 277 279 344 **390**	5
BiLSTM-FL	52 64 58 220 239 246 253 240 221 147 148 169 158 202 243 245 256 277 278 279 281 304 305 307 308 312 313 316 289 286 285 290 291 292 344 345 **390**	37
BiLSTM-FL(UNITE)	239 240 243 245 277 279 285 344 **390**	9
ProFL	348 349 344 345 **390**	5
ProFL(UNITE)	344 **390**	2
DeepRL4FL	243 245 246 344 345 **390**	6
DeepRL4FL(UNITE)	243 245 344 **390**	4

Although Table 15 shows the ranking list, we cannot visually see the locations of the ranking list and call relationship in the faulty program. Thus, Fig 7 shows visual FL results of the 12 FL approaches with and without UNITE. In Fig 7, for each of the 12 FL approaches, we use the same symbol (*i.e.,* a colored rectangle with a solid or dotted line) to mark the locations (*i.e.,* the statements) of the ranking list (*i.e.,* the one in Table 15) in the faulty program. In addition, when there is a call between different functions, we use an arrow with a solid line to denote the call action. As a reminder, for those FL approaches with the same ranking list, we use the symbol to represent their ranking list, *e.g.,* GP02, GP02, ER1', Dstar, Ochiai, ProFL. Taking ER5 as an example, Table 15 shows that its ranking list has 40 statements, meaning that the faulty statement is ranked 40th. Therefore, in Fig 7, the 40 statements are marked with the same yellow and solid line rectangle, showing the distribution of the locations of the ranking list in the program via using ER5. As shown in Fig 7, after applying UNITE, we can visually see that the searching scope of locating the fault is significantly reduced.

Thus, based on the FL results of the qualitative example, we can safely conclude that UNITE significantly improves FL effectiveness. For enabling the qualitative analysis on other programs, we include the complete information about UNITE, faulty locations and the subject programs in the online package[11].

5.5 Threats to Validity

Threats to internal validity. Threats to internal validity relate to potential errors in our implementation. First, one potential threat to validity is the potential errors in the implementation of UNITE and 12 baselines. To mitigate the threat, for eight SFL techniques, we implement them based on the widely used SFL source code GZoltar [12]; for four DLFL techniques, we use and enhance the source code from the previous studies to implement them on source

[11] https://github.com/oy-sarah/UNITE/tree/master/subjectPorgrams.
[12] https://gzoltar.com/

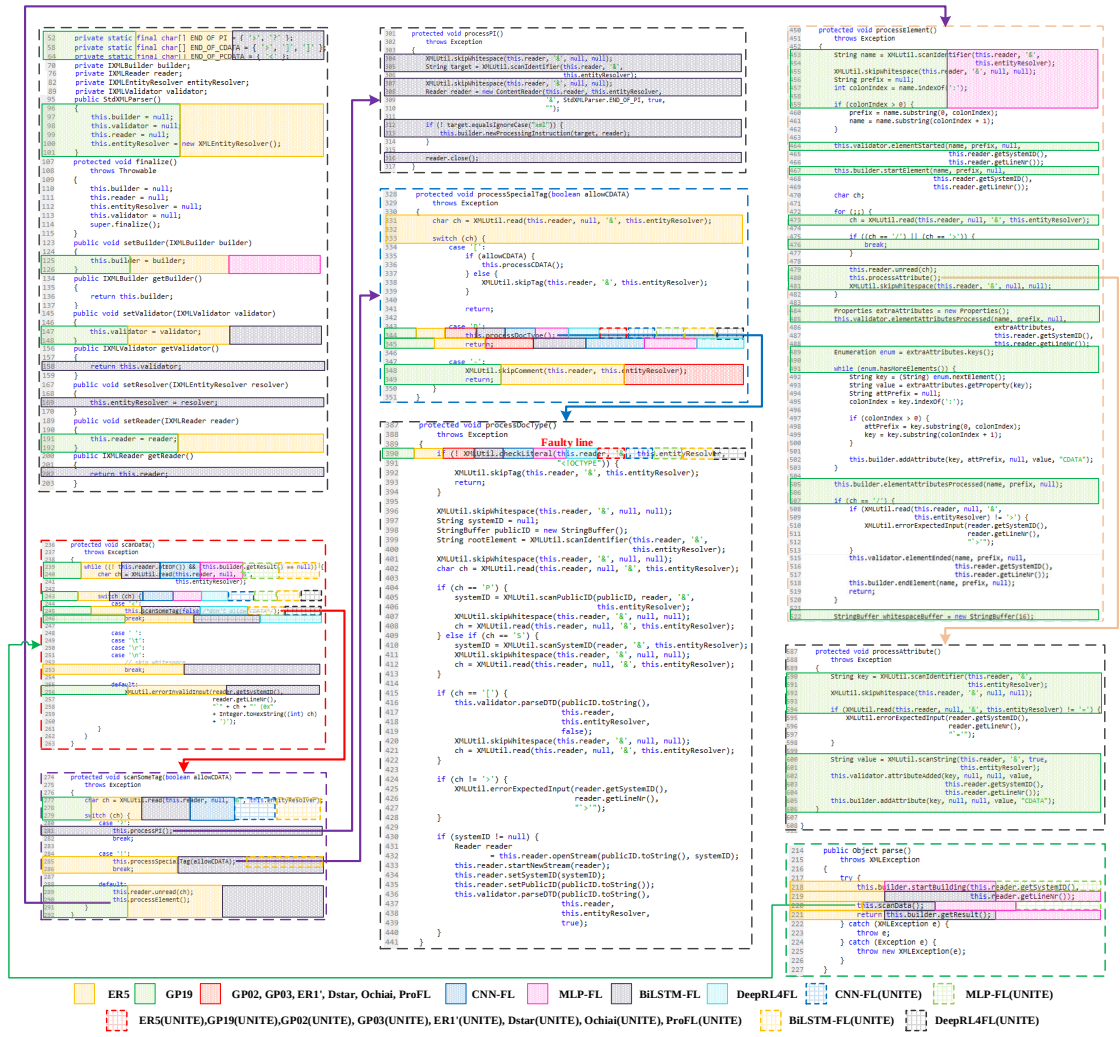

Fig. 7. Visual FL results of the qualitative example.

code [66, 67]. We also double-checked the implementation and fully tested our code, but there could be errors that we did not notice.

Threats to external validity. Threats to external validity relate to generalizability of our results. We use FL techniques using neural networks (*i.e.*, MLP-FL, CNN-FL, BiLSTM-FL and DeepRL4FL), whose outputs are not stable, meaning that the localization results are not the same through different training times. That drawback is caused by characteristic of deep learning technology. To make the results more reliable, we follow the convention strategy by repeating the experiments ten times and using the average score as the experimental results.

Another threat to external validity is the subject programs used for our experiments. Our subject programs are commonly used in the field of software debugging, which are all from the real-life development. However, the experimental results may not apply to all programs because there are still many unknown and complicated factors in realistic debugging that could affect the experiment results. For example, in our approach, a specific failing test case is needed for the obtain of an influential slice to exclude irrelevant statements for a smaller inspecting scope. However, such a choice strategy is suitable for single-fault scenarios since the chosen failing test case can only reveal its own root cause. Consequently, if there are more than one fault contained in a program, the remaining faults will be ignored, *i.e.,* our approach can be affected by multiple-faults scenarios. Specifically, for multiple faults, we have two typical problems. The one is that dynamic information is partially related to multiple faults, *i.e.,* a failing test case only executes part of all the faulty statements of multiple faults. Dynamic FL approaches including UNITE cannot obtain the dynamic information of unexecuted faulty statements, and thus it is difficult for dynamic approaches to be effective at locating those faulty statements not executed by the failing test case. The other one is that multiple faults have complicated effect (*e.g.,* fault interference and coupling effect [6, 8, 64]), which is still difficult to be accurately analyzed. Dynamic slicing used by our approach UNITE also suffer from this problem, and may miss part of all the faulty statements of multiple faults. Consequently, UNITE is ineffective at locating those faulty statements of multiple faults missed by dynamic slicing. To alleviate the problem, we may leverage clustering technology (*e.g.,* [14]) to alleviate the effect by transforming the context of multiple faults into that of single faults. Thus, it is worthwhile to incrementally extend our study to more applications (*e.g.*, multiple-faults programs) to seek additional insights.

Threats to construct validity. Threats to construct validity relate to the suitability of our evaluation. We adopt the widely used metrics (*i.e., TopN*, *MAR*, *MFR* and *RImp*) to evaluate UNITE. According to the extensive use of the measurements, the threat is acceptably mitigated.

6 CONCLUSION AND FUTURE WORK

In this paper, we propose UNITE: an inf**l**U**U**ential co**N**text-Gu**I**ded **T**race r**E**presentation, to represent the trace from both global and local contexts with influential semantics for effective FL. UNITE embodies two key ideas: (1) not only local context but also global context is useful for FL trace representation. (2) program dependencies are potential for upgrading 'occurrence' semantics. To implement the two key ideas UNITE uses the widely-used weighting capability of information retrieval to combine global and local contexts, and further leverages program slicing to incorporate influence semantics into the trace representation through program dependencies. We apply UNITE to 12 state-of-the-art FL techniques and conduct large-scale experiments on 22 benchmark programs. The results show that UNITE significantly improves 12 FL techniques, *e.g.,* the average relative improvement for the most important Top-N metric [17], *i.e.,* Top-1, Top-3, Top-5 and Top-10, achieves 35.58%, 119.90%, 47.43% and 50.66%, respectively.

In the future, we plan to design sophisticated weighting functions for a further optimization on global and local contexts. We also plan to compose influence semantics with other solutions proposed in the literature to improve FL effectiveness (*e.g.,* feature selection).

ACKNOWLEDGMENTS

This work is partially supported by the National Natural Science Foundation of China (Nos. 62272072 and 62002034), the National Defense Basic Scientific Research Project (No. WDZC20205500308) and the Major Key Project of PCL (No. PCL2021A06).

REFERENCES

[1] Hervé Abdi. 2007. The Bonferonni and Šidák Corrections for Multiple Comparisons. *Encyclopedia of measurement and statistics* 3 (2007), 103–107.

[2] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. *ACM SIGPlan Notices* 25, 6 (1990), 246–256.

[3] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 1–10.

[4] L. C Briand, Y Labiche, and Xuetao Liu. 2007. Using Machine Learning to Support Debugging with Tarantula. In *The IEEE International Symposium on Software Reliability*. 137–146.

[5] Gregory W. Corder and Dale I. Foreman. 2010. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Vol. 78. International Statistical Review. 451–452 pages.

[6] Vidroha Debroy and W Eric Wong. 2009. Insights on fault interference for programs with multiple bugs. In *the 20th International Symposium on Software Reliability Engineering*. IEEE, 165–174.

[7] Vidroha Debroy, W. Eric Wong, Xiaofeng Xu, and Byoungju Choi. 2010. A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques. In *International Conference on Quality Software*. 13–22.

[8] Chunrong Fang, Yang Feng, Qingkai Shi, Zicong Liu, Shuying Li, and Baowen Xu. 2017. Fault Interference and Coupling Effect.. In *SEKE*. 501–506.

[9] R. Ranganath H. Lee, R. Grosse and A.Y. Ng. 2009. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th Annual International Conference on Machine Learning*. 609–616.

[10] Simon Heiden, Lars Grunske, Timo Kehrer, Fabian Keller, Andre Van Hoorn, Antonio Filieri, and David Lo. 2019. An evaluation of pure spectrum-based fault localization techniques for large-scale software systems. *Software: Practice and Experience* 49, 8 (2019), 1197–1224.

[11] Jie Lee Hua, Lee Naish, and Kotagiri Ramamohanarao. 2010. Effective Software Bug Localization Using Spectral Frequency Weighting Function. In *IEEE Computer Software and Applications Conference*.

[12] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*. 298–309.

[13] James A. Jones. 2004. Fault Localization Using Visualization of Test Information. In *International Conference on Software Engineering, 2004. ICSE 2004. Proceedings*. 54–56.

[14] James A Jones, James F Bowring, and Mary Jean Harrold. 2007. Debugging in parallel. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis (ISSTA 2007)*. ACM, 16–26.

[15] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE)*. 273–282.

[16] Fabian Keller, Lars Grunske, Simon Heiden, Antonio Filieri, Andre van Hoorn, and David Lo. 2017. A critical evaluation of spectrum-based fault localization techniques on a large-scale software system. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 114–125.

[17] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 165–176.

[18] Tien Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information retrieval and spectrum based bug localization: better together. In *Joint Meeting on Foundations of Software Engineering*. 579–590.

[19] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering* 38, 1 (2011), 54–72.

[20] Yann Lecun, Yoshua Bengio, and G eoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436.

[21] Yan Lei, Xiaoguang Mao, Ziying Dai, and Chengsong Wang. 2012. Effective Statistical Fault Localization Using Program Slices. In *Computer Software and Applications Conference*. 1–10.

[22] Yan Lei, Xiaoguang Mao, Min Zhang, Jingan Ren, and Yinhua Jiang. 2017. Toward understanding information models of fault localization: Elaborate is not always better. In *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, Vol. 1. IEEE, 57–66.

[23] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. 2018. How test suites impact fault localisation starting from the size. *IET Software* 12, 3 (2018), 190–205.

[24] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 169–180.

[25] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the Acm on Programming Languages* 1, OOPSLA (2017), 1–30.

[26] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *International Conference on Software Engineering, 2021. ICSE 2021.*

[27] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. 2019. TBar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 31–42.

[28] Yiling Lou, Ali Ghanbari, Xia Li, Lingming Zhang, Haotian Zhang, Dan Hao, and Lu Zhang. 2020. Can automated program repair refine fault localization? a unified debugging approach. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 75–87.

[29] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An extensible java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 468–478.

[30] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *Journal of Systems and Software* 89, 1 (2014), 51–62.

[31] Wes Masri and Rawad Abou Assi. 2010. Cleansing Test Suites from Coincidental Correctness to Enhance Fault Localization. In *Third International Conference on Software Testing*.

[32] Wes Masri and Rawad Abou Assi. 2014. Prevalence of Coincidental Correctness and Mitigation of its Impact on Fault Localization. *Acm Transactions on Software Engineering and Methodology* 23, 1 (2014), 1–28.

[33] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*. 153–162.

[34] J.J. DiCarlo N. Pinto, D. Doukhan and D.D. Cox. 2009. A high-throughput screening approach to discovering good forms of biologically inspired visual representation. In *PLoS computational biology*. vol.5.

[35] Lee Naish and Hua. 2011. A model for spectra-based software diagnosis. *Acm Transactions on Software Engineering and Methodology* 20, 3 (2011), 1–32.

[36] Mike Papadakis and Yves Le Traon. 2012. Using Mutants to Locate "Unknown" Faults. In *IEEE Fifth International Conference on Software Testing, Verification and Validation*. 691–700.

[37] Mike Papadakis and Yves Le Traon. 2015. *Metallaxis-FL: Mutation-based fault localization.* John Wiley and Sons Ltd. 605–628 pages.

[38] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *International Symposium on Software Testing and Analysis*. 199–209.

[39] Spencer Pearson, Jose Campos, and Just. 2017. Evaluating and Improving Fault Localization. In *International Conference on Software Engineering*.

[40] A. J. C. van Gemund R. Abreu, P. Zoeteweij. 2006. An evaluation of similarity coeffcients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. 39–46.

[41] Anand Rajaraman and Jeffrey David Ullman. 2011. Mining of massive datasets: Data mining. *Min. Massive Datasets* (2011), 1–17.

[42] Abreu Rui, Peter Zoeteweij, and Arjan J. C. Van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Ieee/acm International Conference on Automated Software Engineering*. 88–99.

[43] Abreu Rui, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[44] Cong Ying Shi, X. U. ChaoJun, and Xiao Jiang Yang. 2009. Study of TFIDF algorithm. *Journal of Computer Applications* (2009).

[45] Ting Shu, Tiantian Ye, Zuohua Ding, and Jinsong Xia. 2016. Fault localization based on statement frequency. *Information Sciences* 360 (2016), 43–56.

[46] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 273–283. https://doi.org/10.1145/3092703.3092717

[47] Ezekiel Soremekun, Lukas Kirschner, Marcel Böhme, and Andreas Zeller. 2021. Locating faults with program slicing: an empirical analysis. *Empirical Software Engineering* 26, 3 (2021), 1–45.

[48] Chengnian Sun and Siau Cheng Khoo. 2013. Mining succinct predicated bug signatures. In *Joint Meeting on Foundations of Software Engineering*. 576–586.

[49] S. C. Turaga, J. F. Murray, V Jain, F Roth, M Helmstaedter, K Briggman, W Denk, and H. S. Seung. 2010. Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural Computation* 22, 2 (2010), 511–538.

[50] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[51] Xinming Wang, Shing Chi Cheung, Wing Kwong Chan, and Zhenyu Zhang. 2009. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In *IEEE International Conference on Software Engineering*.

[52] W. Eric Wong, Vidroha Debroy, and Byoungju Choi. 2010. *A family of code coverage-based heuristics for effective fault localization.* Elsevier Science Inc. 188–208 pages.

[53] W. Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2012. Effective Software Fault Localization Using an RBF Neural Network. *IEEE Transactions on Reliability* 61, 1 (2012), 149–169.

[54] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software Fault Localization Using DStar (D*). In *IEEE Sixth International Conference on Software Security and Reliability*. 21–30.

[55] W. Eric Wong, Ruizhi Gao, Yihao Li, Abreu Rui, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

[56] W. ERIC WONG and YU QI. 2009. BP Neural Network-based Effective Fault Localization. *International Journal of Software Engineering and Knowledge Engineering* 19, 04 (2009), 573–597.

[57] W. Eric Wong, Yu Qi, Lei Zhao, and Kai Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. International*. 449–456.

[58] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems. In *Proceedings of the IEEE*

International Conference on Software Maintenance and Evolution (ICSME). 267–278.

[59] Xiaoyuan Xie, Tsong Yueh Chen, Fei Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *Acm Transactions on Software Engineering and Methodology* 22, 4 (2013), 31.

[60] Xiaoyuan Xie, Fei Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. *Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation.* Springer Berlin Heidelberg. 224–238 pages.

[61] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2 (2005), 1–36.

[62] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Transactions on Software Engineering* 43, 1 (2016), 34–55.

[63] Lei Yan, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. 2018. How Test Suites Impact Fault Localization Starting from the Size. *Iet Software* 12, 3 (2018), 190–205.

[64] Abubakar Zakari, Sai Peck Lee, Rui Abreu, Babiker Hussien Ahmed, and Rasheed Abubakar Rasheed. 2020. Multiple fault localization of software programs: A systematic literature review. *Information and Software Technology* 124 (2020), 106312.

[65] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Xi Chang, Jianxin Xue, and Qingyu Xiong. 2020. Fault Localization Approach Using Term Frequency and Inverse Document Frequency. *Journal of Software* 31, 11 (2020), 132–144.

[66] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*. IEEE, 445–455.

[67] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Junhao Wen. 2020. Improving Deep-Learning-based Fault Localization with Resampling. *Journal of Software: Evolution and Process* (2020), 1–22.

[68] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Ling Xu, and Xiaohong Zhang. 2021. A study of effectiveness of deep learning in locating real faults. *Information and Software Technology* 131 (2021), 106486.

[69] Zhuo Zhang, Yan Lei, Qingping Tan, Xiaoguang Mao, Ping Zeng, and Xi Chang. 2017. Deep Learning-Based Fault Localization with Contextual Information. *Ieice Transactions on Information and Systems* E100.D, 12 (2017), 3027–3031.

[70] Zhuo Zhang, Yan Lei, Jianjun Xu, Xiaoguang Mao, and Xi Chang. 2019. TFIDF-FL: Localizing Faults Using Term Frequency-Inverse Document Frequency and Deep Learning. *IEICE Transactions on Information and Systems* 102, 9 (2019), 1860–1864.

[71] Wei Zheng, Desheng Hu, and Jing Wang. 2016. Fault Localization Analysis Based on Deep Neural Network. *Mathematical Problems in Engineering,2016,(2016-4-24)* 2016 (2016), 1–11.

[72] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* (2019), 1–1.