PAPER

Applying Association Analysis to Dynamic Slicing Based Fault Localization

Heling CAO[†], **Shujuan JIANG**[†a)], **Xiaolin JU**[†,††], **Yanmei ZHANG**[†], ***Nonmembers*, *and* Guan YUAN**[†], ***Member***

SUMMARY Fault localization is a necessary process of locating faults in buggy programs. This paper proposes a novel approach using dynamic slicing and association analysis to improve the effectiveness of fault localization. Our approach utilizes dynamic slicing to generate a reduced candidate set to narrow the range of faults, and introduces association analysis to mine the relationship between the statements in the execution traces and the test results. In addition, we develop a prototype tool DSFL to implement our approach. Furthermore, we perform a set of empirical studies with 12 Java programs to evaluate the effectiveness of the proposed approach. The experimental results show that our approach is more effective than the compared approaches.

key words: *dynamic slicing, fault localization, association analysis, execution trace*

1. Introduction

Program debugging is generally estimated to consume 50% or more of the total development and maintenance effort [1], [2]. Fault localization is one of the most difficult and time-consuming tasks in program debugging [4], [7]. To alleviate this problem, developers usually use debugging tools to manually place breakpoints, observe the error states, and then backtrack to fix faults. Obviously, this process is quite laborious and time-consuming. To reduce the cost of program debugging and improve software quality, researchers have investigated methods of helping to automate the process of fault localization [3]–[7].

Automated fault localization approaches usually try to narrow the range of faults significantly. Delta debugging [2] compares memory states of the passed and failed runs and narrows the differences between these states to a subset of suspicious variables through manipulating memory states of the two runs. Dynamic slicing, which is firstly proposed by Korel and Laski [8], narrows the search space to a set of statements which influence an output variable. Zhang *et al.* [3] proposed dynamic slicing for fault localization, Wen [12] utilized dynamic slicing with statistical analysis to locate faults.

Coverage-based fault localization approaches are another effective ways to locate faults. In general, they utilize statistical analysis to compute the suspiciousness of the statements being faulty, and generate a ranking list of statements in terms of the suspiciousness. Tarantula is a typical coverage-based fault localization technique, which is proposed by Jones *et al.* [10]. Ochiai is proposed by Abreu *et al.* [16] to augment the Tarantula technique. Naish1 [11] and Wong1 [17] are the maximal risk evaluation formulas that are theoretically proved by Xie *et al.* [18]. One limitation of these approaches is that they only focus on the suspiciousness of the statements and do not take into account association information within the program, leading to the inaccuracy of fault localization. In fact, there is the relationship between the suspicious code and the failed test results. Association analysis [9] is introduced to obtain some correlations between program statements and the corresponding test results from a large number of program executions.

In this paper, we propose a novel fault localization approach FLDA (Fault Localization based on Dynamic slicing and Association analysis) to locate faults effectively. First, dynamic slicing is utilized to narrow the range of the faults. Second, we propose a ranking strategy RSA (a Ranking Strategy based on Association analysis), which is applied to reduce the cost of fault localization. Association analysis is introduced to find the correlations between the statements in the execution traces and the failed test results, and then a ranking strategy is introduced to generate a more reasonable ranking list. Finally, developers examine suspicious statements guided by this ranking list. To evaluate the effectiveness of the proposed approach, we perform empirical studies with 12 Java programs. The results indicate that our approach has a significant improvement over the compared ones and reduces almost 2.26% to 8.47% of the cost on average.

The main contributions of this paper can be summarized as follows:

- A novel fault localization approach combining dynamic slicing with association analysis is proposed to improve the effectiveness of fault localization.
- A new ranking strategy for the statements based on association analysis is designed to generate a ranking list.
- An evaluation for the effectiveness of our approach is performed with 6 fault localization approaches across 12 Java programs. The experimental results show that our approach can outperform the compared ones.

The rest of this paper is organized as follows. Section 2 presents a motivating example for fault localization. Section 3 presents our approach and its detailed description.

Manuscript received November 28, 2013.
Manuscript revised March 20, 2014.

[†]The authors are with School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, China.
[††]The author is with School of Computer Science and Technology, Nantong University, Nantong, China.
a) E-mail: shjjiang@cumt.edu.cn

DOI: 10.1587/transinf.E97.D.2057

Section 4 illustrates the experimental study and its data analysis. Section 5 presents some related work and Sect. 6 draws conclusions and discusses some potential future work.

2. Motivation

In this section, the program `function()` with one faulty statement s_2 in Fig. 1 is taken as an example. As shown in Table 1, there are eight test cases: t_1, t_2 and t_7 are failed test cases, t_3, t_4, t_5, t_6 and t_8 are passed test cases. The symbol '√' denotes that a statement is executed. The first column shows the number of statements. The second column shows coverage information under eight test cases. The columns to the right of coverage information column give the suspiciousness and ranking by Tarantula [10] (i.e., formula (1)), and ranking by RSA, respectively, for statements based on coverage information. The fifth column shows dynamic slices. The columns to the right of dynamic slices column give the suspiciousness and ranking by Tarantula, and ranking by RSA, respectively, based on dynamic slices under eight test cases. Row `True/Fail` shows the test results (i.e., True (T) or Fail (F)), row `fault rank.` illustrates the number of the examined statements (e.g., "4–9" means of examining 4 up to 9 statements) to locate faults and row `loc.cost (%)` presents the cost of fault localization.

```
     function () {
        int x, y, z, m, ret ;
s1      read (x, y, z);
s2      x = x/y;   // correct: x = x/y − 1;
s3      m = y;
s4      if (x > 1)  {
s5           m = x − 3;
s6           x = x/y − 1;
        }
s7      if (m > 0)
s8           ret = x + z;
        else
s9           ret = y − z;
s10     return ret;
     }
```

Fig. 1 A motivating example.

$$Tarantula(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{failed(s)}{totalfailed} + \frac{passed(s)}{totalpassed}} \tag{1}$$

In formula (1), *failed(s)* and *passed(s)* denote the number of the statement s, which was executed by failed and passed test cases, respectively, while *totalfailed* and *totalpassed* denote the number of failed and passed test cases. For example, statement s_1 has a suspiciousness of 0.5 because it is executed by three failed test cases out of a total of three failed test cases (i.e., The ratio is 1.), and it is executed by five passed test cases out of a total of five passed test cases (i.e., The ratio is 1.). We obtain $1/(1 + 1)$, or 0.5 by formula (1). Similarly, the maximum suspiciousness is equal to 0.62 corresponding to the ranking of 1.

The statements are examined to locate the fault in descending order of the suspiciousness. Based on coverage information, 4 statements (i.e., s_8, s_5, s_6, s_2) need to be examined in the best case and 9 statements (i.e., s_8, s_5, s_6, s_1, s_3, s_4, s_7, s_{10}, s_2) need to be examined in the worst case by Tarantula until faulty statement s_2 is located, because statements s_8, s_5, s_6 have higher suspiciousness than s_2 and statements s_1, s_3, s_4, s_7, s_{10} have the same suspiciousness as s_2. If we construct dynamic slices and calculate suspiciousness, Tarantula successfully locates faulty statements s_2 only by examining 3 to 7 statements (row `fault rank.`). This motivates us to adopt dynamic slices to improve the accuracy of fault localization.

However, dynamic slices [8] are a set of statements relevant to the fault, and the elements in it are treated equally. When developers examine the statements in dynamic slices, what is the order? To address this problem, we use association analysis to explore some correlations between statements in the execution traces and the failed test results, and we use a new ranking strategy based on association analysis to improve the ranking of the fault (see Sect. 3.3). From Table 1, we can observe that our approach (RSA) needs to examine 1 to 6 statements based on coverage information and needs to examine 1 to 5 statements based on dynamic slices to locate the faults. The cost of our approach is lower than that of Tarantula. Therefore, in this paper, association

Table 1 Comparisons of fault localization on a motivating example.

program	coverage information								Tarantula		RSA	dynamic slices								Tarantula		RSA
	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	sus.	rank.	rank.	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	sus.	rank.	rank.
s_1	√	√	√	√	√	√	√	√	0.50	4	1	√	√	√	√	√	√	√	√	0.50	3	1
s_2 (faulty)	√	√	√	√	√	√	√	√	0.50	4	1	√	√	√	√	√	√	√	√	0.50	3	1
s_3	√	√	√	√	√	√	√	√	0.50	4	1	√				√	√			0.45	8	8
s_4	√	√	√	√	√	√	√	√	0.50	4	1	√	√	√	√	√	√	√	√	0.50	3	1
s_5		√	√	√			√	√	0.53	2	8		√	√	√			√	√	0.53	2	7
s_6		√	√	√			√	√	0.53	2	8			√				√	√	0.45	8	9
s_7	√	√	√	√	√	√	√	√	0.50	4	1	√	√	√	√	√	√	√	√	0.50	3	1
s_8	√		√				√	√	0.62	1	7	√		√				√	√	0.62	1	6
s_9		√		√	√	√			0.36	10	10		√		√	√	√			0.36	10	10
s_{10}	√	√	√	√	√	√	√	√	0.50	4	1	√	√	√	√	√	√	√	√	0.50	3	1
True/Fail	F	F	T	T	T	T	F	T				F	F	T	T	T	T	F	T			
fault rank.										4–9	1–6										3–7	1–5
loc. cost (%)										40–90	10–60										30–70	10–50

analysis and ranking strategy will be implemented to reduce the cost of fault localization.

3. Fault Localization Guided by Dynamic Slicing and Association Analysis

In this section, we present the details of FLDA. First, dynamic slices are calculated according to data dependency and control dependency of program entities in order to reduce the search domain of faults. Second, association analysis is applied to explore the correlations between the statements in the execution traces and the failed test results, and then the ranking strategy we presented is applied to generate a more effective ranking list of the statements.

3.1 Dynamic Slicing

Dynamic slicing, which is calculated with respect to each interest point (called a slice criterion) in the source code, is introduced to analyze a program and constructed by dynamic dependency during the process of program execution. When analyzing the execution trace under a particular test case, we only need to analyze the statements in dynamic slices.

Definition 1 (Execution trace): Let I be a given input, the execution trace T of a program run is a statement sequence. Each element of trace T is given a unique *timestamp* denoted as ${}^{timestamp}s$, which represents the serial number of the statement executed. According to the test results, they are divided into failed execution traces T_f and passed execution traces T_p.

Definition 2 (Dynamic slicing criterion): Dynamic slicing criterion is a triple C = <I, s, V>, where I is a program input, s is an interest point corresponding to a statement and V is a subset of variables in statement s.

Definition 3 (Dynamic slicing): Given a slice criterion C = <I, s, V>, dynamic slicing is a set of statements, which directly or indirectly affect variable V in statement s, where I is a program input [8].

Dynamic slicing is computed to get a subset of the executed statements which actually affect the value of the variable at that point. In this paper, dynamic slice algorithm is forward, which means that the dynamic slice is created as soon as this statement has been executed.

Algorithm 1 illustrates forward computation of dynamic slices algorithm. The algorithm takes current execution instance ${}^{i}s$ as input, and takes $slice[v]$ as output. Lines 1–4 calculate dynamic slice of variable u in $Use[s]$. $Use[s]$ and $Def[s]$ denote the dynamic sets of variables which are defined and used by statement s, respectively. Similarly, Lines 5–9 calculate dynamic slice of variable v in $Def[s]$. $CD[s]$ denotes the set of predicate statements on which s is control dependent. Dynamic control dependency of ${}^{i}s$ is recorded as dcd. The algorithm returns $slice[v]$ of defined variable v as output. After execution of instance ${}^{i}s$, the dynamic slice of defined variables in statement s is computed, including the following statements: statements that belong to the latest dynamic slices of variables used by ${}^{i}s$; statements that belong to the dynamic slice of predicate on which ${}^{i}s$ is dynamic control dependent; and the statement s itself. If variable v is defined in statement s, the latest dynamic slice for variable v is $slice[v]$.

Now we apply Algorithm 1 on the example program in Fig. 1 to illustrate the dynamic slice. A dynamic slice of the program is computed for the slice criterion C = <($x = 4, y = 2, z = 1$), s_{10}, ret> in Table 2.

Algorithm 1 Forward Computation of Dynamic Slices

```
Require:
    ${}^{i}s$, which means current execution instance
Ensure:
    $slice[v],\ v \in Def[s]$
 1: $slicetemp = \{i\}$;
 2: for each $u$ in $Use[s]$ do
 3:     $slicetemp = slicetemp \cup slice[u]$;
 4: end for
 5: $dcd$ = the statement in $CD[s]$ which has the maximum timestamp value;
 6: $slicetemp = slicetemp \cup slice[dcd]$;
 7: for each $v$ in $Def[s]$ do
 8:     $slice[v] = slicetemp$;
 9: end for
10: return $slive[v]$;
```

3.2 Association Analysis

In this section, association analysis searches for the correlations between the statements in dynamic slices and the test results. After that, a ranking strategy is applied to prioritize the statements.

The value of discriminating function $f(s)$ is 1 when statement s is only executed by failed executions, otherwise its value is 0. That is to say, if $f(s)$ value of a statement is equal to 1, the higher priority is assigned to the statement. The definition of $f(s)$ is as follows:

$$f(s) = \begin{cases} 1 & s \in T_f \wedge s \notin T_p \\ 0 & otherwise \end{cases} \tag{2}$$

Definition 4 (Association Rule): An association rule is an implication expression of the form $A \Rightarrow B$, where $A \subset D$, $B \subset D$, and $A \cap B = \emptyset$. A is called the antecedent of the rule and B is called the consequent of the rule [9]. The rule

Table 2 Forward computation of dynamic slices under test case t_2.

${}^{i}s$	Def[s]	Use[s]	CD[s]	dynamic slices
${}^{1}s_1$	{x,y,z}	$\emptyset$	$\emptyset$	$\{s_1\}$
${}^{2}s_2$	{x}	{x,y}	$\emptyset$	$\{s_1, s_2\}$
${}^{3}s_3$	{m}	{y}	$\emptyset$	$\{s_1, s_3\}$
${}^{4}s_4$	$\emptyset$	{x}	$\emptyset$	$\{s_1, s_2, s_4\}$
${}^{5}s_5$	{m}	{x}	${}^{4}s_4$	$\{s_1, s_2, s_4, s_5\}$
${}^{6}s_6$	{x}	{x,y}	${}^{4}s_4$	$\{s_1, s_2, s_4, s_6\}$
${}^{7}s_7$	$\emptyset$	{m}	$\emptyset$	$\{s_1, s_2, s_4, s_5, s_7\}$
${}^{8}s_9$	{ret}	{y,z}	${}^{7}s_7$	$\{s_1, s_2, s_4, s_5, s_7, s_9\}$
${}^{9}s_{10}$	$\emptyset$	{ret}	$\emptyset$	$\{s_1, s_2, s_4, s_5, s_7, s_9, s_{10}\}$

is measured by support and confidence.

Let $D = \{D_1, D_2, \cdots, D_m\}$ be the set of all items and $T = \{T_1, T_2, \cdots, T_n\}$ be the set of all transactions. Each transaction T_i contains a subset of items chosen from D. Let A is item set, and transaction T contains A if and only if $A \subseteq T$. In the context of fault localization, a program execution is viewed as a transaction, and all program executions construct transactions T. The statements in the execution traces are referred as the antecedent of the rule, and the failed test result as the consequent of the rule.

Definition 5 (Support): Support determines how often a rule is applicable to a given data set, which is the probability of transactions that simultaneously contain A and B, where a rule $A \Rightarrow B$ is recorded as support($A \Rightarrow B$), which is the probability of P($A \cup B$) [9]. The formal definition of support is

$$support(A \Rightarrow B) = P(A \cup B) \tag{3}$$

where $min\_support = \frac{1}{N}$

In the context of fault localization, symbol A represents the statements in a program, and symbol B represents the failed test results. The threshold value of support equals to $\frac{1}{N}$ (N = $|T_p| + |T_f|$), because at least one failed test case is necessary to reveal the presence of the faults.

Definition 6 (Confidence): Confidence determines how frequently item set B appears in the transactions that contain A [9], which is the conditional probability of P(B | A). The formal definition of confidence is

$$confidence(A \Rightarrow B) = P(B \mid A) = \frac{support(A \cup B)}{support(A)} \tag{4}$$

where $min\_confidence = \frac{1}{N}$

The threshold value of confidence is equal to $\frac{1}{N}$, because at least one failed test case is necessary to reveal the presence of the faults, and the faulty statements are covered by all execution traces in extreme cases.

Most existing coverage-based fault localization approaches compute the suspiciousness of statements based on the following three assumptions [10], [13], [15], [17].

- The suspiciousness of the statement is positive to the number of the statement that is covered by the failed executions.
- The suspiciousness of the statement is inverse to the number of the statement that is not covered by the failed executions.
- The suspiciousness of the statement is inverse to the number of the statement that is covered by the passed executions.

In light of the three assumptions, the three-level ranking strategy is proposed as follows:

(1) Rank the statements according to their $f(s)$ values. The larger its $f(s)$ value is, the higher the priority is. That is to say, we set the highest priority to the statement when its $f(s)$ value is equal to 1.
(2) Rank the statements according to their support values, when multiple statements have the same $f(s)$ values. The larger its support value is, the higher the priority is. That is to say, the more the statement is covered by the failed executions, the higher priority assigned to this statement.
(3) Rank the statements according to their confidence values, when multiple statements have the same support values. That is to say, the less the statement is covered by the passed executions, the higher priority assigned to this statement.

Algorithm 2 presents association analysis and ranking strategy algorithm. The algorithm takes *slice* (i.e. a set of dynamic slices of all tests), *min_support* and *min_confidence* as inputs. The algorithm returns a sorted statement sequence S' as output. Line 1 initializes a list S'. Lines 2–6 compute $f(s)$, support and confidence by formula (2), (3) and (4), respectively. The symbol "Fail" in line 4 represents the failed test result. Association analysis is applied to mine the relationship between the statements in the execution traces and the test results by setting *min_support* and *min_confidence* and then the ranking strategy is applied to generate a ranking list (lines 7–17). Line 18 returns a statement sequence S' sorted by ranking strategy. Additionally, function *comp*() in line 10 is used to compare statements s_1 with s_2. The return value of *comp*() is true when satisfying the three conditions as follows:

(1) $f(s_1) < f(s_2)$;
(2) $f(s_1) = f(s_2) \wedge support(s_1) < support(s_2)$;
(3) $f(s_1) = f(s_2) \wedge support(s_1) = support(s_2)$ $\wedge\, confidence(s_1) < confidence(s_2)$.

Algorithm 2 Association Analysis and Ranking Strategy

```
Require:
    slice
    min_support
    min_confidence
Ensure:
    S'
 1: list $S'$ = InitInput( );
 2: for each $s$ in Slice do
 3:     computef($s$);
 4:     $support = P(s \cup Fail)$;
 5:     $confidence = P(s \cup Fail)/P(s)$;
 6: end for
 7: for $i = 1; i < S'.size; i++$ do
 8:     for $j = 1; j < S'.size - i; j++$ do
 9:         if $support \geq min\_support \&\& condence \geq min\_confidence$ then
10:             if $comp(S'[j], S'[j+1]) == True$ then
11:                 $swap(S'[j], S'[j+1])$;
12:             end if
13:         else
14:             $delete(s)$;
15:         end if
16:     end for
17: end for
18: return $S'$;
```

3.3 Fault Localization

We developed a prototype tool DSFL (Dynamic Slicing Fault Locator) to implement the proposed approach. The Soot analysis framework is applied to obtain the data flow and control flow information. And the construction of the execution traces is achieved by instrumenting the code through Java Debug Interface (JDI). JDI is a high-level debugging interface in Java platform architecture and provides the ability of accessing and controlling the target virtual machine for a debugger.

We use the program `function()` in Fig. 1 to illustrate the proposed approach. First, dynamic slices are calculated as follows. The program `function()` is executed on input I. The dynamic slices shown in Table 3 are obtained for a slice criterion $<I, s_{10}, \{t\}>$ when the last statement was executed. Second, association analysis is implemented on the basis of dynamic slices and then ranking strategy is implemented to generate a ranking list of the statements. Association analysis is applied to explore the correlations between the statements in the execution traces and test results by setting *min_support* and *min_confidence* in Algorithm 2. Before association analysis, we illustrate how their metrics are computed. For example, statement s_2 is executed by both failed and passed test cases, so the value of $f(s_2)$ is equal to 0 by formula (2); statement s_2 is executed by three failed test cases and a total of test cases is eight, so $support(s_2 \Rightarrow Fail) = 3/8$ by formula (3); and statement s_2 is executed by all test cases, $support(s_2) = 1$, so $confidence(s_2 \Rightarrow Fail) = support(s_2 \cup Fail)/support(s_2) = 0.375$ by formula (4).

Table 4 gives the details of the three-level ranking strategy as follows:

(1) The first level ranking is shown in row 2. The statement is examined firstly when its $f(s)$ value is equal to 1. Multiple statements are further prioritized according to their support values when they have the same $f(s)$ values. The statements of the program in Table 1 are executed by both failed and passed test cases, so the $f(s)$ value of each statement is equal to 0 by formula (1).

(2) The second level ranking is shown in row 3. The statements, such as s_1, s_2, s_4, s_7 and s_{10}, should be examined firstly because they have the larger support values than the other statements. Next, statement s_5 or s_8 will be examined. Multiple statements are further prioritized according to their confidence values when they have the same support values.

(3) The third level ranking is shown in row 4. The statements should be examined firstly when they have the larger confidence values. For example, statement s_8 is examined before statement s_5 because it has the larger confidence values than statement s_5.

Finally, if multiple statements have the same support and confidence values, it is possible to be prioritized according to their serial numbers. Therefore, developers can examine the source code in order $s_1 \rightarrow s_2 \rightarrow s_4 \rightarrow s_7 \rightarrow s_{10} \rightarrow s_8 \rightarrow s_5 \rightarrow s_3 \rightarrow s_6 \rightarrow s_9$. Similar is the computation process of the ranking based on coverage information.

If we prioritize the statements by the reverse above ranking strategy, or if we cancel the first level ranking when the faulty statement is only executed by failed test cases, the accuracy of fault localization will decrease.

Now we utilize the program `function()` in Fig. 1 to illustrate the effectiveness of our approach. The comparative experiments are conducted between Tarantula and RSA in Table 1. Based on coverage information, we need to examine 40% to 90% of the code using Tarantula, whereas, we need to examine 10% to 60% of the code using RSA. Similarly, based on dynamic slices, we need to examine 30% to 70% of the code using Tarantula, whereas, we need to examine 10% to 50% of the code using RSA. The experimental results show that RSA outperforms Tarantula in both cases.

In the next section, we will perform empirical studies to further evaluate the effectiveness of our approach and the compared ones using complex and large scale programs.

Table 3 Dynamic slices for program `function()`.

test case	input	test result	dynamic slices
t_1	{1,2,1}	Fail	$s_1, s_2, s_3, s_4, s_8, s_9, s_{10}$
t_2	{4,2,1}	Fail	$s_1, s_2, s_4, s_5, s_7, s_9, s_{10}$
t_3	{9,1,1}	True	$s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_{10}$
t_4	{−4,−2,1}	True	$s_1, s_2, s_4, s_5, s_7, s_9, s_{10}$
t_5	{4,−2,1}	True	$s_1, s_2, s_3, s_4, s_7, s_9, s_{10}$
t_6	{2,−1,1}	True	$s_1, s_2, s_3, s_4, s_7, s_9, s_{10}$
t_7	{8,2,2}	Fail	$s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_{10}$
t_8	{5,1,1}	True	$s_1, s_2, s_4, s_5, s_6, s_7, s_8, s_{10}$

4. Experimental Study

To evaluate the effectiveness of FLDA, we implemented it in a prototype tool DSFL and performed empirical studies across 12 Java programs. In our study, we want to address the following research questions:

- RQ1: Can FLDA outperform the compared slice-based fault localization approaches?
- RQ2: Can FLDA perform better than the compared coverage-based fault localization approaches?

Table 4 A ranking strategy based on association analysis.

	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9	s_{10}
$f(s)$	0	0	0	0	0	0	0	0	0	0
support	3/8	3/8	1/8	3/8	2/8	1/8	3/8	2/8	1/8	3/8
confidence	0.375	0.375	0.333	0.375	0.4	0.333	0.375	0.5	0.25	0.375
ranking	1	1	8	1	7	9	1	6	10	1

Table 5 Subjects, test cases and faults.

subject	description	LOC	faults	test cases
print_tokens	lexical analyzer	478	7	4130
print_tokens2	lexical analyzer	410	5	4115
schedule	priority scheduler	290	9	2650
schedule2	priority scheduler	317	8	2710
tcas	collision avoidance	131	31	1608
tot_info	information measure	283	10	1052
NanoXML v1	XML parser	3497	7	214
NanoXML v2	XML parser	4009	7	214
NanoXML v3	XML parser	4608	8	216
XML-security v1	XML encryption	21613	7	92
XML-security v2	XML encryption	22318	5	94
XML-security v3	XML encryption	19895	7	84

- RQ3: Can FLDA improve the effectiveness of fault localization significantly?

4.1 Subjects for Analysis

Table 5 lists the subjects we evaluated and provides the description, lines of code (LOC) excluding the number of non-blank and non-commented lines of Java code, the number of faults and the number of test cases. The former 6 subject programs were the Siemens suite, which Santelices *et al*. [15] had translated from C version to Java version[†]. Specially, the Siemens suite was chosen for evaluation because it was widely used in many empirical studies [10], [12], [14]. Three releases of NanoXML and XML-security, which are large scale program obtaining from Software Infrastructure Repository (SIR) with faults and test cases[††], were studied in our experiment. We excluded the faulty versions of the programs that caused no failure because at least one failed test case is necessary in order to trigger the failure. In total, 111 faulty versions of the programs were examined under the single-fault scenario.

4.2 Experimental Design

To verify the effectiveness of dynamic slicing and association analysis, we evaluate association analysis based on dynamic slicing (i.e., FLDA) or association analysis based on coverage information (i.e., RSA), respectively. FLDA performs better than RSA. Therefore, the approach FLDA we proposed combines association analysis with dynamic slicing. We evaluate the effectiveness of FLDA in terms of the average cost, EXAM score and Wilcoxon signed-rank test and evaluate the effectiveness of RSA only in terms of the average cost. In addition, because multiple statements possibly have the same suspiciousness, the faulty statement is examined first in the best case and last in the worst case. Without generality, we adopt the average between the best and the worst.

[†]http://www3.nd.edu/˜rsanteli/subjects/
[††]http://sir.unl.edu/portal/index.php

4.3 Data Analysis

Table 6 illustrates the average fault localization cost (the ratio of the cumulative number of examined statements to the executable statements) for all faulty versions on each subject. We group all subjects into two and choose the first 6 subjects for one group and the remaining 6 programs for the other because of the different characteristics of programs. Table 7 shows the average cost and the standard deviation of the cost for groups. The table shows both (i) the average cost of the two groups and (ii) the average cost of all subjects.

To compare two approaches M and N for the effectiveness, the reduced cost of M over N is computed by subtracting the average cost of M from the average cost of N. For example, in the fifth row of Table 7, the average cost of FLDA is 4.80% and the average cost of Tarantula is 12.54%, then the reduced cost of FLDA over Tarantula is 7.74%, which means developers would examine 7.74% fewer code if they used FLDA. We next present the effectiveness comparisons between the fault localization approaches.

(1) Comparing RSA with coverage-based fault localization approaches (i.e., Tarantula [10], Ochiai [16], Naish1 [11] and Wong1 [17]). From Table 6, for each subject, we can observe that RSA is more effective than the compared approaches except for Ochiai, Naish1 for print_tokens. From Table 7, for group GR1 and GR2, we can observe that the average cost of RSA is always less than that of the compared approaches, respectively. For all subjects, it shows that the average cost of RSA (i.e., 8.71%) is less than that of the compared approaches, and that the reduced cost of RSA over Tarantula, Ochiai, Naish1 and Wong1 are 3.83%, 2.69%, 2.56% and 4.56%, respectively. In addition, the standard deviation of RSA is much smaller than that of the compared approaches, whether GR1, GR2 or all subjects, therefore, RSA is more stable in the effectiveness than the compared approaches.

(2) Comparing FLDA with coverage-based and slice-based fault localization approaches. From Table 6, for each subject, we can observe that FLDA is more effective than the compared approaches except for FS [3] and JHSA [12]. The average cost of FLDA is higher than that of JHSA for XML-security v2, and higher than that of FS for print_tokens and XML-security v2. From Table 7, for group GR1 and GR2, we can observe that FLDA is more effective than the compared approaches. For all subjects, it shows that the average cost of FLDA (i.e., 4.80%) is less than that of the compared approaches, and that the reduced cost of FLDA over Tarantula, Ochiai, Naish1, Wong1, FS and JHSA are 7.74%, 6.60%, 6.47%, 8.47%, 4.30% and 2.26%, respectively. In addition, the standard deviation of FLDA is the smallest one in that of the compared approaches, therefore, FLDA is more stable in

Table 6 Comparisons of the average cost of fault localization for each subject.

group	subject	FLDA	FS	JHSA	RSA	Tarantula	Ochiai	Naish1	Wong1
GR1	print_tokens	5.32%	16.50%	8.37%	11.98%	18.05%	8.73%	10.73%	14.97%
	print_tokens2	8.78%	12.98%	14.34%	9.37%	24.39%	25.12%	24.54%	25.17%
	schedule	4.56%	14.56%	8.58%	4.98%	6.70%	7.16%	5.36%	9.66%
	schedule2	8.52%	5.01%	9.58%	25.63%	31.43%	31.62%	31.31%	36.99%
	tcas	9.36%	20.93%	13.54%	16.60%	23.52%	20.91%	22.97%	22.97%
	tot_info	8.83%	19.79%	14.06%	17.70%	19.43%	17.81%	18.73%	23.82%
GR2	NanoXML v1	4.46%	3.90%	5.72%	5.11%	7.19%	6.74%	5.72%	7.97%
	NanoXML v2	1.10%	2.50%	1.58%	3.61%	4.57%	3.95%	3.66%	4.66%
	NanoXML v3	1.57%	6.00%	2.03%	3.53%	6.84%	6.63%	4.51%	4.38%
	XML-security v1	1.42%	3.90%	2.51%	1.52%	2.38%	2.25%	1.79%	2.31%
	XML-security v2	1.20%	0.60%	0.65%	1.48%	2.20%	2.15%	2.20%	2.24%
	XML-security v3	2.43%	2.50%	3.77%	3.01%	3.73%	3.71%	3.73%	4.05%

Table 7 Average cost and standard deviation for the group.

group	statistics	FLDA	FS	JHSA	RSA	Tarantula	Ochiai	Naish1	Wong1
GR1	average cost	7.56%	14.96%	11.41%	14.38%	20.59%	18.56%	18.94%	22.26%
	standard dev	1.88%	5.23%	2.61%	6.60%	7.54%	8.62%	8.68%	8.57%
GR2	average cost	2.03%	3.23%	2.71%	3.04%	4.49%	4.24%	3.60%	4.27%
	standard dev	1.17%	1.66%	1.64%	1.26%	1.96%	1.86%	1.33%	1.91%
All Subjects	average cost	4.80%	9.10%	7.06%	8.71%	12.54%	11.40%	11.27%	13.27%
	standard dev	3.18%	7.03%	4.87%	7.40%	9.76%	9.50%	9.87%	10.93%

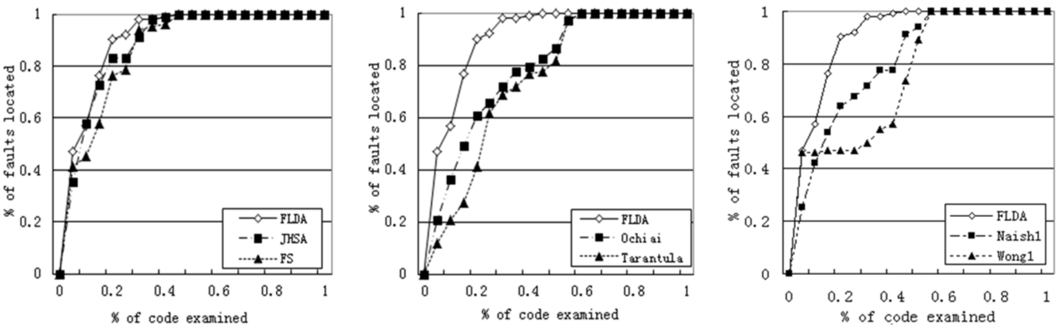


Fig. 2 EXAM-based comparisons on all subjects.

the effectiveness than the compared approaches.

(3) Comparing FLDA with RSA. From Tables 6, 7, for each subject or groups, we can observe that the average cost of FLDA is lower than that of RSA. The reduced cost of FLDA over RSA mainly contributes to the introduction of dynamic slices which remove fault-irrelevant statements. For all subjects, the reduced cost of FLDA over RSA is 3.91%.

The EXAM score, which is defined as the percentage of the executable code that must be examined before the faults are identified [4], [16], is an effective evaluation measurement. Approach A performs better than approach B, which means that approach A can locate more faults than approach B when examining the same ratio of the code. Figure 2 shows the effectiveness comparisons on the EXAM score between our approach and the compared ones. The y-axis of each subplot represents the percentage of the faults that are located, while the x-axis represents the percentage of the code that is examined. In Fig. 2 (a), when examining 20% of the code, FLDA can locate 90% of the faults, while JHSA can locate 83% of the faults and FS can locate 75% of the faults. Similarly, in Fig. 2 (b), when examining 20% of the code, Ochiai can locate 55% of the faults and Tarantula can locate 41% of the faults. And in Fig. 2 (c), when examining 20% of the code, Naish1 can locate 63% of the faults and Wong1 can locate 47% of the faults. From Fig. 2 we observe that FLDA is the most effective of the compared approaches.

Additionally, the Wilcoxon signed-rank test is also conducted to identify whether FLDA is more effective than the other approaches significantly. It is a nonparametric test, which can test the difference for paired data. To identify whether there is a significant difference between FLDA and each compared approach in Table 6, the Wilcoxon signed-rank test is performed as follows:

Step 1: present a null hypothesis that FLDA requires to examine more statements than the other techniques. When the null hypothesis is rejected, an alternative hypothesis needs to be accepted. The null hypothesis is H_0: $\mu 1 = \mu 2$, while the alternative hypothesis is H_1: $\mu 1 > \mu 2$.

Step 2: set the level of significance $\alpha = 0.05$.

Step 3: specify the test statistic T, which is that FLDA achieves better experimental results than another approach.

Step 4: supposing X_i and Y_i are the paired experimental results of FLDA and another approach. Define $m = m_+ + m_-$, where m_+ and m_- denotes positive rank and negative rank with $X_i < Y_i$ and $X_i > Y_i$, respectively. Then the binomial probability p-value is calculated.

Step 5: if the p-value is less than or equal to α, the null hypothesis has to be rejected. In our experiment, it can be found that the p-values of all test are less than 0.05, so we reject the null hypothesis and consider that the difference between FLDA and the other techniques is significant.

From the above empirical results, for all subjects, faults and test cases in our experiment, we can answer the three proposed research questions as follows:

- For RQ1, we can draw a conclusion that FLDA can improve the performance of fault localization to some extent. Comparing with FS and JHSA, FLDA can reduce the average cost by 2.26% to 4.30%.
- For RQ2, we can draw a conclusion that FLDA outperforms the compared coverage-based approaches. Comparing with Tarantula, Ochiai, Naish1 and Wong1, FLDA can reduce the average cost by 6.47% to 8.47%.
- For RQ3, we can draw a conclusion that the difference of the experimental results between FLDA and the compared approaches is significant at 95% confidence level by the Wilcoxon signed-rank test.

4.4 Threats to Validity

The external validity of this experiment lies in generalizing our approach. Specifically, the results obtained using 12 Java programs in size from 131 to 22318 cannot be generalized to other diversity programs. However, all subject programs in our experiment are widely used in fault localization researches (i.e., [10], [13], [15]), furthermore, NanoXML and XML-security are the representative of real-world software, this increases the reliability of our experiment to some extent.

The internal validity of this experiment lies in the precision of dynamic slices obtained by our tool. However, we utilized forward computation of dynamic slices algorithm similar to the algorithm proposed by Beszedes *et al.* [25]. Dynamic slices are used to narrow the search space, and the results show that dynamic slices we obtained are effective in fault localization.

5. Related Work

There are many coverage-based fault localization approaches. Jones *et al.* [10] proposed the Tarantula fault localization approach with the suspiciousness of the statements indicating the probability of the statements being faulty. Abreu *et al.* [16] presented the similarity coefficient named Ochiai that originated in the molecular biology field to compute the suspiciousness of statements. Xie *et al.* [18] provided a theoretical proof on the effectiveness of the 30 suspiciousness formulas and proved that five out of the 30 suspiciousness formulas were maximal formulas. Xu *et al.* [28] proposed a novel similarity coefficient with a noise reduction to compute the suspiciousness of statements for improving the effectiveness of fault localization.

Dynamic slices of multiple failed and passed test executions were exploited by Pan and Spafford [5], who proposed a family of heuristics to compare the dynamic slices of failed and passed test executions for identifying possible faulty statements. Zhang *et al.* [3] computed data slices, full slices and relevant slices respectively. Their experimental results indicated that full slices and relevant slices had better results than data slices in fault localization. In their follow-up study, they [6] proposed another fault localization approach based on dynamic slices by computing from multiple points (i.e. erroneous value, critical predicate). Yu *et al.* [7] calculated approximate dynamic slices of test executions and then calculated the suspiciousness of each statement using Tarantula based on the number of dynamic slices of failed and passed test executions. Mao *et al.* [24] presented approximate dynamic backward slices to balance the size and precision of the slices and then statistically computed the suspiciousness. In the previous work [19], we presented a fault localization approach based on hybrid spectrum of full slices and execution slices and computed the suspiciousness of statements by a maximal formula we designed. Hofer *et al.* [26] combined spectrum-based fault localization and slicing-hitting-set-computation to improve the ranking of the faults. In their follow-up study, they [27] combined spectrum-based fault localization and artificial intelligence techniques to improve diagnostic accuracy.

Similar to our work, Wotawa [29] proposed a fault localization approach using dynamic slicing and model-based diagnosis to improve the effectiveness. In addition, Wen [12] combined dynamic slicing with statistical analysis to locate the faults. Comparing with our approach, Wotawa's approach used model diagnosis for computing the probabilities of states to be faulty, Wen's approach utilized a suspicious metric similar to Tarantula to compute the suspiciousness of statements, whereas, our approach utilized a ranking strategy based on association analysis to generate a ranking list for locating faults.

Association analysis, a branch of data mining, was firstly introduced by Agrawal [9]. One of the earliest applications of association analysis in fault localization was in [20], where Denmat and Ducasse utilized the support and confidence metrics to filter fault-related statements in execution traces. Wong *et al.* [21] presented a fault-localization technique using N-gram analysis and association analysis. They constructed the actual execution sequence of length N and mined the correlations between the subsequences and the test results. Cellier *et al.* [22] discussed the fault-failure correlations using formal concept and association rule to locate faults. Different from previous research, we introduce a novel application of association analysis to locate faults. We applied it to the statements and do not construct a sequence of length N to avoid expensive time overhead. In addition, our approach combined association analysis with a ranking strategy to generate a ranking list.

6. Conclusions and Future Work

In this paper, we presented a novel approach, based on dynamic slicing and association analysis, to improve the effectiveness of fault localization. We implemented the proposed approach in a prototype tool and designed a set of empirical studies for evaluating the effectiveness of FLDA with coverage-based and slice-based fault localization approaches. The experimental results indicate that FLDA is more effective than the compared approaches, approximately reducing the fault localization cost from 2.26% to 8.47% on average.

In future work, we plan to apply our approach to locate multiple faults. We also plan to apply another data mining technique, such as clustering, classification, etc., to locate faults.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work is supported by the Fundamental Research Funds for the Central Universities under Grant No. 2012LWB40.

References

[1] J.S. Collofello and S.N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," J. Systems and Software, vol.9, no.3, pp.191–195, 1989.

[2] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," IEEE Trans. Softw. Eng., vol.28, no.2, pp.183–200, 2002.

[3] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," Proc. International Symposium on Automated Analysis-Driven Debugging, pp.33–42, 2005.

[4] W.E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*), " Proc. International Conference on Software Security and Reliability," pp.21–30, June 2012.

[5] H. Pan and E.H. Spafford, "Heuristics for automatic localization of software faults," Technical Report, SERC-TR-116-P, Purdue University, 1992.

[6] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," Software: Practice and Experience, vol.37, pp.935–961, 2007.

[7] R. Yu, L. Zhao, L. Wang, and X. Yin, "Statistical fault localization via semi-dynamic program slicing," Proc. International Conference on Trust, Security and Privacy in Computing and Communications, pp.695–700, 2011.

[8] B. Korel and J. Laski, "Dynamic slicing," Inf. Process. Lett., vol.29, no.3, pp.155–163, 1988.

[9] P.N. Tan, M. Steinbach, and V. Kumar, Introduction to data mining, Posts & Telecom Press, 2006.

[10] J.A. Jones, M.J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," Proc. International Conference on Software Engineering, pp.467–477, 2002.

[11] L. Naish, H.J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," ACM Trans. Software Engineering and Methodology, vol.20, no.3, pp.11–43, 2011.

[12] W. Wen, "Software fault localization based on program slicing spectrum," Proc. International Conference on Software Engineering, pp.1511–1514, 2012.

[13] W.E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," J. Systems and Software, vol.83, no.2, pp.188–208, 2010.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and control flow-based test adequacy criteria," Proc. International Conference on Software Engineering, pp.191–200, 1994.

[15] R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold, "Lightweight fault-localization using multiple coverage types," Proc. International Conference on Software Engineering, pp.56–66, 2009.

[16] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "On the accuracy of spectrum based fault localization," Proc. Testing: Academic and Industrial Conference, Practice and Research Techniques, pp.89–98, Sept. 2007.

[17] W.E. Wong, Y. Qi, L. Zhao, and K.Y. Cai, "Effective fault localization using code coverage," Proc. Annual International Conference on Computer Software and Applications, vol.1, pp.449–456, 2007.

[18] X. Xie, T.Y. Chen, F.C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," ACM Trans. Software Engineering and Methodology, vol.22, no.4, pp.31–40, 2013.

[19] X. Ju, S. Jiang, X. Chen, X. Wang, Y. Zhang, and H. Cao, "HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices," J. Systems and Software, vol.90, pp.3–17, April 2014.

[20] T. Denmat, M. Ducasse, and O. Ridoux, "Data mining and cross-checking of execution traces: A re-interpretation of Jones, Harrold and Stasko test information," Proc. International Conference on Automated Software Engineering, pp.396–399, 2005.

[21] S. Nessa, M. Abedin, W.E. Wong, L. Khan, and Y. Qi, "Software fault localization using N-gram analysis," Wireless Algorithms, Systems, and Applications, Lecture Notes in Computer Science, pp.548–559, 2008.

[22] P. Cellier, S. Ducasse, S. Ferre, and O. Ridoux, "Formal concept analysis enhances fault localization in software," Proc. International Conference on Formal Concept Analysis, pp.273–288, 2008.

[23] N. DiGiuseppe and J.A. Jones, "On the influence of multiple faults on coverage-based fault localization," Proc. International Symposium on Software Testing and Analysis, pp.210–220, 2011.

[24] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," J. Systems and Software, vol.89, pp.51–62, 2014.

[25] A. Beszedes, T. Gergely, Z.M. Szabo, J. Csirik, and T. Gyimothy, "Dynamic slicing method for maintenance of large C programs," Proc. European Conference on Software Maintenance and Reengineering, pp.105–113, March 2001.

[26] B. Hofer and F. Wotawa, "Spectrum enhanced dynamic slicing for better fault localization," Proc. European Conference on Artificial Intelligence, pp.420–425, 2012.

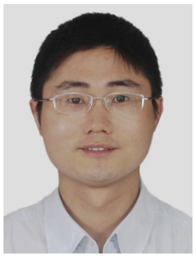
[27] B. Hofer, F. Wotawa, and R. Abreu, "AI for the win: Improving spectrum-based fault localization," ACM SIGSOFT Software Engineering Notes, vol.37, no.6, pp.1–8, Nov. 2012.

[28] J. Xu, W.K. Chan, Z. Zhang, T.H. Tse, and S. Li, "A dynamic fault localization technique with noise reduction for Java programs," Proc. International Conference on Quality Software, pp.11–20, July 2011.

[29] F. Wotawa, "Fault localization based on dynamic slicing and hitting-set computation," Proc. International Conference on Quality Software, pp.161–170, July 2010.

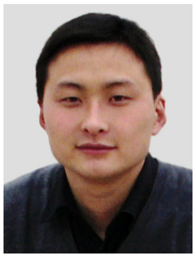
Heling Cao is currently a Ph.D. candidate in school of computer science and technology, China University of Mining and Technology. The main areas of interest are software engineering and data mining.

Shujuan Jiang is a Professor at School of Computer Science and Technology, China University of Mining and Technology where she teaches graduate and undergraduate courses of software engineering, compiling technique and conducts research in these domains. She had 6 month of experience as a visiting scholar in College of Computing, Georgia Institute of Technology, Atlanta, Georgia, U.S.A. in the year 2008–2009.



Xiaolin Ju is currently a Ph.D. candidate in school of computer science and technology, China University of Mining and Technology. The main area of interest is Software analysis and testing.

Yanmei Zhang received the Ph.D. degree from School of Computer Science and Technology, China University of Mining and Technology in 2012. The main areas of interest are software engineering and software quality.



Guan Yuan received the Ph.D. degree from School of Computer Science and Technology, China University of Mining and Technology. The main areas of interest are data mining and software engineering.