International Journal of Signal Processing, Image Processing and Pattern Recognition
Vol. 10, No. 8 (2017), pp.63-74
http//dx.doi.org/10.14257/ijsip.2017.10.8.06

Research of Multiple Fault Localization Based on Cluster Analysis of Program Failures

Xiaoan Bao[1], Yusen Wang[1], Junyan Qian[2], Zijian Xiong[1], Na Zhang[1*] and Chenghai Yu[1]

[1]The institute of software of Zhejiang Sci-Tech University, Hangzhou 310027, China
[2]Guilin University of Electronic Technology, Guilin 541004, China
zhangna@zstu.edu.cn

Abstract

Locating faults is one of the most expensive and time-consuming components of debugging process. Fault localization technique based on mining associations analyzes the dependencies between application code and narrow down the location of faults. However, the efficiency of this technology will decrease with the increase of the number of faults. This paper presents a new fault location technique based on cluster analysis of program failures. Failures are categorized into different failure classes using cluster analysis method, and the failures are caused by one and only one fault in each class. We also study characteristics of a set of statements covered by failed executions, which are due to the same fault. According to failure classes' difference, we describe a target association algorithm and a corresponding way to examine code. Empirical studies based on SIR benchmarks indicate that, for the subject we studied, our technique has higher efficiency than the popular Tarantula, Ochiai and Jaccard techniques in multiple-fault programs, and can be implemented in effective space and time complexity.

Keywords*: multi-fault location, cluster analysis, failure classes, target association, software test.*

1. Introduction

It is commonly recognized that debugging software is an expensive and mostly manual process. Of all debugging activities, locating faults is a very resource-consuming task, including the time and the cost. Due to its high cost, any improvement in the process of finding faults is able to greatly decrease the cost of debugging. Depending on such realizations, a variety of fault localization techniques have been proposed recently, and each of which aims to guide programmers to the locations of faults in one way or another [1-12].

Reducing the search scopes is the main approach of fault localization technique to improve efficiency. In [1], Cleve and Zeller reported a program state-based debugging technology, called *Cause Transitions*, to recognize the locations and times, in which a cause of failure changes from one variable to another. This is an expansion of their earlier work with delta debugging [2-3]. An algorithm named cts is proposed to quickly locate cause transitions in a program execution. One of the potential problems of the cause transition technique is that the cost of it is relatively high; there may not only

[1] Na Zhang is the corresponding author.

exist thousands of states in a program execution, also delta debugging at each matching point needs additional test runs to narrow down the causes.

Traditional program slicing technique can be roughly classified as static [4] and dynamic [5]. The technique gets slices that are associated with the fault to reduce the searching area by analyzing the data dependences and control dependences in programs. However, the time complexity and space complexity of conventional program slicing technique are quite big, and program slicing is conservative, so they are inefficient.

Current spectrum-based approaches for fault localization assess the suspiciousness of program entities to locate faults [6-10]. Effectively, the coverage information cannot reflect the complicated control- and data-dependency relationships, and simplify the execution spectra. On this basis, mining associations-based fault localization technique [12] has been shown to have better diagnostic performance than spectrum-based approaches. However, with the increase of the number of bugs, failed executions caused by different faults would affect the suspiciousness of the same statement. And the effect of fault localization will reduce, when blocks with high suspiciousness span through multiple function bodies.

This paper presents the details of our multiple fault localization technique with a description of cluster analysis. This paper also presents the results of empirical studies that evaluate the technique to determine whether it helps to locate faults in the program. We performed the study on C programs with a number of faulty versions, every version containing from two to five known faults. In particular, we make the following contributions:

1. A presentation of a new technique for cluster-analysis-based multiple fault localization (CA-MFL), which provides a new approach for categorizing the failures caused by different faults. The clustering analysis method helps in locating two or more faults in a program by illuminating likely faulty statements which belong to the same kind of failures.
2. We describe a target association algorithm and a corresponding way to review codes, according to the failure classes' differences, which are not restricted to a single way and can maintain the effectiveness of fault localization.
3. We systematically evaluate the effectiveness of cluster-based multiple fault localization on the SIR benchmarks [13] under the same setting as previous studies. Four existing fault localization techniques are compared with our technique in this study, which demonstrates the superior accuracy achieved by our technique in fault localization.

2. Preliminaries

In this section we introduce program spectra, and present a detailed example which illustrates the advantage of modeling cluster analysis.

In this paper we use the following terminology. A *failure* is an event that occurs when delivered service deviates from correct service. An *error* is a system state that may cause a failure. A *fault* is the cause of an error in the system. In our discussion, faults are bugs in program code, and failures occur when the actual output for a given input deviates from the expected output for that input.

2.1. Program Spectra

A program spectrum is a collection of data that provides a specific view on dynamic behaviors of software. Numerous different forms of program spectra exist [14], in this section we work with so-called basic block spectra. A basic block spectrum includes a block of code in a program, which cannot be branched into or out of. This data is collected at run-time, and typically consists of a lot of counters or flags for the different parts of a program.

To facilitate basic block spectra, given a test suite T for a software system S and a test case t in T, this technique requires two types of information about the execution of S with t: pass/fail results and code coverage. Test case t passes if the actual output for an execution of S with t is the same as the expected output for t. If it is, the test case t is *a passing case* and the execution is *a passing execution*; otherwise, t fails, the test case t is *a failing case* and the execution trace is *a failing execution* or *a failure* in short.

The spectra of M constitutes a binary matrix, whose rows correspond to N different blocks of the program (see Table 1). Where $x_{i,j} = \{0,1\}$, and $1 \le i \le n$, $1 \le j \le m$. Besides, $x_{i,j} = 1$ indicates the block B_i was touched in the execution of test case t; otherwise, $x_{i,j} = 0$. The fail/pass results of the test cases are listed as “F” or “P”.

Table 1. The Ingredients of Fault Localization

Basic Block	Test Cases (T)						Suspiciousness
	F			P			
	t_1	…	t_s	t_{s+1}	…	t_m	
b_1	$x_{1,1}$	…	$x_{1,s}$	$x_{1,s+1}$	…	$x_{1,m}$	p_1
b_2	$x_{2,1}$	…	$x_{2,s}$	$x_{2,s+1}$	…	$x_{2,m}$	p_2
⋮	⋮		⋮	⋮		⋮	⋮
b_n	$x_{n,1}$	…	$x_{n,s}$	$x_{n,s+1}$	…	$x_{n,m}$	p_n

Many similarity coefficients exist. As an example, below are three different similarity coefficients, namely the Tarantula coefficients, used in the Tarantula fault localization tool [6], the Jaccard coefficients which is used by the Pinpoint tool [7-8], and the Ochiai coefficients, taken from the molecular biology domain [7-8]:

$$\text{Tarantula}(b) = \frac{n_{11}(b)/\big(n_{11}(b)+n_{01}(b)\big)}{n_{11}(b)/\big(n_{11}(b)+n_{01}(b)\big) + n_{10}(b)/\big(n_{10}(b)+n_{00}(b)\big)} \tag{1}$$

$$\text{Jaccard}(b) = \frac{n_{11}(b)}{n_{11}(b)+n_{01}(b)+n_{10}(b)} \tag{2}$$

$$\text{Ochiai}(b) = \frac{n_{11}(b)}{\sqrt{\big(n_{11}(b)+n_{01}(b)\big)*\big(n_{11}(b)+n_{10}(b)\big)}} \tag{3}$$

Where$n_{uv}(b)=|\{a|x_{ab}=u \wedge r_a=v\}|$, and $u,v \in \{0,1\}$. Besides, $x_{ab}=u$ indicates whether block b was involved $(u=1)$ in the execution of run a or not $(u=0)$. Likewise, $r_a=v$ indicates whether one run a was faulty $(v=1)$ or not $(v=0)$.

Under the assumption that a high similarity to the faulty block indicates a high probability that the corresponding statements of the software cause the detected bugs, the calculated similarity coefficients rank the parts of the program in regard to their likelihood of containing the faults.

2.2. Motivating Example

In this section, we present a simple example of a C program that is designed to illustrate the process of finding and localizing faults. Consider the erroneous program for determining the max of the four input values, as shown in Figure 1. Faults lie on basic blocks 3 and 6 in this program which has been divided into 8 basic blocks. There, the value m, representing the max, should be assigned the value of z rather than the value of a or b.

```
          int m, x, y, z, w;
b1        Read(x, y, z, w);
          if(x > y)
          -----------------------------------
              m = x;
b2            if(m < z)
          -----------------------------------
              m = x;      /*fault, b3' is the revised block;*/
b3            if(m < w)
            /*b3'  m = z;
              if(m < w ) */
          -----------------------------------
b4            m = w;
          -----------------------------------
          else
b5            m = y;
              if(m < z)
          -----------------------------------
               m = y;   /*fault, b6' is the revised block;*/
b6             if(m < w)
            /*b6'  m = z;
               if(m < w ) */
          -----------------------------------
b7             m = w;
          -----------------------------------
b8         print(m);
```

Figure 1. The Instance Containing Two Faults

2.2.1. single-fault localization: Suppose that the program has a single fault which occurred on basic block 6. The similarity coefficients have been calculated using three techniques above, and Table 2 shows the corresponding block spectra. In this figure, blocks that were touched were shown by "1", otherwise blocks were shown by "0".

Table 2. Example of basic Block Spectra-based Single Fault Localization

Basic Blocks	Test Cases (*T*)								Suspiciousness		
	F		P								
	6,6,7,5	6,7,9,3	8,6,9,7	5,4,6,8	7,6,6,4	4,5,6,7	5,7,6,3	7,6,8,8	Tarantula	Jaccard	Ochiai
b_1	1	1	1	1	1	1	1	1	0.50	0.25	0.50
b_2	0	0	1	1	1	0	0	1	0.00	0.00	0.00
$b_{3'}$	0	0	1	1	1	0	0	1	0.00	0.00	0.00
b_4	0	0	0	1	0	0	0	0	0.00	0.00	0.00
b_5	1	1	0	0	0	0	1	1	0.75	0.50	0.71
b_6	1	1	0	0	0	0	1	1	0.75	0.50	0.71
b_7	0	0	0	0	0	0	1	0	0.00	0.00	0.00
b_8	1	1	1	1	1	1	1	1	0.50	0.25	0.50

Eight test cases are supplied to the program as shown in the table: two of the executions produce the incorrect output and six of the executions produce correct output. Each test case is represented by a column in Table 2. The test-case input is at the top of each column; and the fail/pass results from the execution of the test cases, listed as "F" or

"P", respectively, is at the head of the table. For example, the first test case has an input of "6, 6, 7, 5", executes basic blocks b_1,b_6, and b_8, and fails.

Rank the basic blocks of the program with respect to their likelihood of containing the faults according to the calculated similarity coefficients. The second lookup enables you to navigate to the faulty basic block by using mining associations-based fault localization technique, which shows better efficiency.

2.2.2. Multi-fault localization: In this particular example, we suppose that the program has two faults which occurred on basic blocks 3 and 6. The similarity coefficients have been again calculated using three techniques above (see Table 3). In contrast to the example above, the third test case that has an input of "8, 6, 9, 7" fails.

Table 3. Example of basic Block Spectra-based Multiple Fault Localization

Basic Blocks	Test Cases (*T*)								Suspiciousness		
	F			P							
	6,6,7,5	6,7,9,3	8,6,9,7	5,4,6,8	7,6,6,4	4,5,6,7	5,7,6,3	7,6,8,8	Tarantula	Jaccard	Ochiai
b_1	1	1	1	1	1	1	1	1	0.50	0.38	0.61
b_2	0	0	1	1	1	0	0	1	0.36	0.17	0.29
b_3	0	0	1	1	1	0	0	1	0.36	0.17	0.29
b_4	0	0	0	1	0	0	0	1	0.00	0.00	0.00
b_5	1	1	0	0	0	1	1	0	0.63	0.40	0.58
b_6	1	1	0	0	0	1	1	0	0.63	0.40	0.58
b_7	0	0	0	0	0	1	0	0	0.00	0.00	0.00
b_8	1	1	1	1	1	1	1	1	0.50	0.38	0.61

The spectrum identifies basic block 6 as the most likely location of the fault: the suspiciousness of basic block 6, calculated respectively by the Tarantula/Jaccard/Ochiai coefficients, rank first and third, especially the suspiciousness of basic block 3 rank second to last. As expected, the effectiveness of the technique declines as the number of faults increases. Users, with the mining associations-based fault localization technique, are still able to locate faulty block 6 pretty quickly according to suspiciousness ranking order. However, it takes more time and expense to locate faulty block 3.

Of course, this example is contrived in many ways: tests are reasonably small, no latent faults have occurred, the structure of the program is simple, etc. However, it serves to illustrate the mining associations-based fault localization technique does not perform very well.

3. Cluster-Analysis-Based Technique

In this section, we describe the multi-fault localization based on cluster analysis approach and the target association algorithm that implement this intuition.

3.1 Problem Settings

Branch-instruction-based partition technique [15] divides a program during runtime into different blocks, and each block reflects the runtime behavior of the program. For example, the sequence of the program state segments (*CALLS,…,RETS*) denotes that the program performs a function call.

The sequence of the program state segments and the code coverage are recorded at each execution. The code coverage of an execution R of a program, called an execution slice, is the set of program statements which are ever executed in R [16]. On account of the execution slice is essentially a set of executed statements, we can use any distance formula to calculate the pairwise distance between two sets. Jaccard distance, proposed by Levandowsky and Winter, is chosen here [17]. Given two nonempty sets A and B, the Jaccard distance between them is

$$\mathrm{D}(A,B)=1-\frac{|A\cap B|}{|A\cup B|} \tag{4}$$

Where $|A|$ denotes the number of elements of the set A. The Jaccard distance is a metric, satisfying the following four properties:

1.Nonnegativity: $\mathrm{D}(A,B)\geq 0$; $\mathrm{D}(A,B)=0$ if and only if $A=B$;
2.Symmetry: $\mathrm{D}(A,B)=\mathrm{D}(B,A)$; and
3.Triangle inequality: $\mathrm{D}(A,C)\leq\mathrm{D}(A,B)+\mathrm{D}(B,C)$.

Note that we use the Jaccard distance to calculate the distance between failures x_i and x_j in this paper, then the failures can be partition into different failure classes by cluster analysis method.

Let $\mathrm{X}=\{x_1,x_2,\ldots,x_n\}$ be a set of n failures caused by m faults $\mathrm{Y}=\{y_1,y_2,\ldots,y_m\}$, where neither m nor Y are known. Assume an unknown oracle function $\Psi:X\longrightarrow Y$, we specify the relationship between X and Y: A failure x_i is due to the fault y_j if and only if

$$\Psi(x_i)=j \tag{5}$$

The oracle function Ψ divides the set of failures Y into m mutually exclusive and collectively exhaustive failure classes $\mathrm{K}=\{K_j\}_{j=1}^{m}$, where

$$K_j=\{x_i|\Psi(x_i)=j,\ \text{for } i=1,2,\ldots,n\} \tag{6}$$

Given a failure x, we use $K_{\Psi\{x\}}$ to denote the failure class which x belongs to and $K_{\Psi\{x\}}$ includes all failures due to the same fault, such as x.

3.2 Methodology Overview

We formulate the main idea of our method in this section and its details are narrated in Section 3.3. With the distance between failing execution slices as data source, partition the failures into different failure classes using cluster analysis method. Specific steps are as follows:

Executing the program component on the input and checking for failures. Then gets a collection of failures, failing execution slices and the sequence of the program state segments, respectively.

1.Calculating the Jaccard distance between two failures, then get an $n\times n$ proximity matrix M

$$\mathrm{M}=\begin{bmatrix} m_{11} & \cdots & m_{1n} \\ \vdots & \ddots & \vdots \\ m_{n1} & \cdots & m_{nn} \end{bmatrix} \tag{7}$$

Where $m_{i,j}$ is the distance between failures x_i and x_j.

2.Dividing failures based on the computed matrix M into different failure classes using cluster analysis method. We choose here k-means algorithm which is one of the simplest unsupervised learning algorithms and gives a better result.

In accordance with the sequence of the program state segments, the number of function calls of different failure classes is not the same. As the number of times functions are called increases, the program may have more than one infected basic blocks or statements. This is mainly because when a fault affects the output, propagation of the fault occurs. We propose the following way to locate faults in the object code, aiming at the failure classes' differences:

1.Compute the similarity coefficient of each basic block/statement, and create a ranking of basic blocks/statements in descending order. According to the sequence of the program state segments, check if the number of function calls is less than θ($\theta=3$ for this paper), go to step 2. If it is not, continue with step 4.

2.Check each basic block in turn for the presence of the faulty block, then go to step 6. Otherwise, go to step 3.

3.Take the basic block as a target, according to the target association algorithm, and create the target association set. Calculate the similarity coefficient of each element of the set, and create a ranking of elements in descending order. Check each element in turn for any faults, and if it is, go to step 6. Otherwise, go to step 2.

4.Check each statement in turn for the presence of the faulty statement, then go to step 6. Otherwise, go to step 5.

5.Take the statement as a target, and create the target association set. Calculate the similarity coefficient of each element of the set and sort them descending order. Check each element in turn for the presence of the faulty statement, then go to step 6. Otherwise, go to step 4.

6.Count the number of basic blocks/lines of code that have been checked.

If we locate faults in a single way, the effect of fault localization will reduce when blocks with high suspiciousness are in multiple function bodies. In order not to influence the effect of fault localization, we check each suspicious statement in turn when the number of function calls is greater than θ.

3.3 Target Association Algorithm

This section presents our algorithm for obtaining the association set whose elements are in line with the target in coverage information. Given a set of coverage vector $T=<t_1,t_2,...,t_r>$, where t_i indicates that the i^{th} test case is executed. The vector set T is partitioned into two disjoint subsets T_p and T_f, corresponding to the passing and failing cases respectively. Besides, $t_i=<e_1,e_2,...,e_s>$, where

$$t_i(e_j)= \begin{cases} 1, & e_j \text{ is involved in the execution of run } t_i; \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

And $\mathrm{e_j}$ indicates that the j^{th} basic block or statement in the object program. Finally, the workflow of our approach of creating the association set, target association algorithm, is detailed in Alg 1.

To illustrate the algorithm, we mark basic blocks/statements and target association set as OT and TAS respectively. And $tas(e)$ is an association set in which e is treated as the target.

Alg 1 Target Association Algorithm

```
Inputs: Ranking basic blocks or statements $OT$, Failing Vector set $T_f$
Output: Target Association set $TAS$
1.   $tas \in TAS$
2.   $tas(e) \leftarrow I$
3.   $TAS \leftarrow \emptyset$
4.   for each $t \in T_f$
5.       for each $e \in OT$
6.           if $t(e)>0$ then
7.              $tas(e) \leftarrow tas(e) \wedge t$
8.              $TAS \leftarrow TAS \cup tas(e)$
9.           end if
10.      end for
11.  end for
12.  return $TAS$
```

4. Empirical Study

For evaluating the performance of our approach we used the well-known Siemens benchmark set, as well as gzip and sed, which could be obtained from SIR [13]. Every single program has a correct version and a set of faulty versions of the same program. Although the faulty may span multiple statements and/or functions, each faulty version contains single fault. For each program a set of inputs was also provided, which was created to test full coverage. Table 4 provides more information about the programs used in our experiments.

For our experiments, we extended the subject programs with program versions where we could activate arbitrary combinations of 2-5 faults. For this purpose, we limited ourselves to a selection of 143 out of the 183 faults, based on criteria such as faults being attributable to a single line of code, to enable unambiguous evaluation.

Table 4. The Subject Programs

Program	Faulty Versions	Number of Lines	Test Cases	Description
sed	6	14,427	370	Textual manipulator
gzip	7	5,680	210	Data compression
Print_tokens	7	539	4,130	Lexical Analyzer
print_tokens2	10	489	4,115	Lexical Analyzer
replace	32	507	5,542	Pattern Recognition
schedule	9	397	2,650	Priority Scheduler
schedule2	10	299	2,710	Priority Scheduler
tcas	41	174	1,608	Altitude Separation
tot_info	23	398	1,052	Information Measure

We now proceed to evaluate our approach (CA-MFL) in the context of multiple faults, using our extended Siemens benchmark set, gzip and sed. We compared our technique with several well-known techniques, such as Tarantula, Ochiai, Jaccard and SBI. For compatibility with previous work in fault localization, we used the percentage of basic blocks or statements that need to be inspected to localize the fault to evaluate the performance of our technique. All measurements except for the four- fault version of print_tokens were averages over 100 versions, or over the maximum number of combinations available, where we verified that all faults are active in at least one failed run.

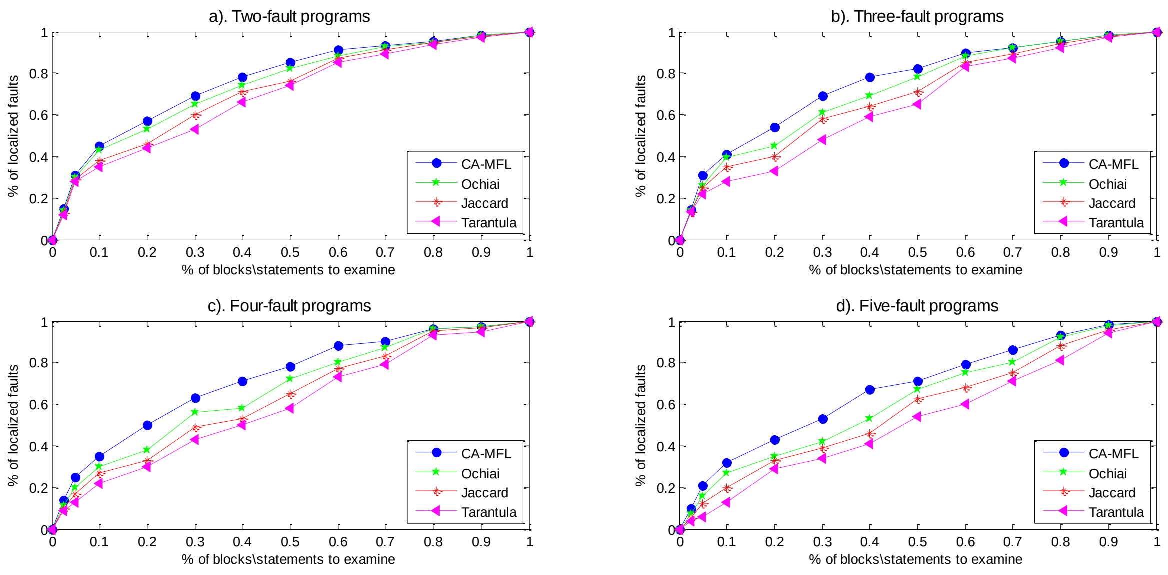

Figure 2. Effectiveness Comparison in two to five-fault programs

Figure 2 plots the percentage of located faults in terms of blocks or statements to examine. From Figure 2 we conclude that CA-MFL is consistently the best performing technique, finding 55% approximately of the faults by examining 20% of the source code. For the same percentage of examined code, using Ochiai leads a developer to find about 40% of the faulty versions, and with Tarantula only 30% is found. Our approach outperforms Ochiai, which is consistently better than Jaccard and Tarantula.

And then we report on the time/space complexity of CA-MFL, compared to other fault localization techniques. The time complexity of the target associate algorithm and ranking components in fault likelihood are $O(T{\cdot}e_s)$ and $O(M{\cdot}\log M)$ respectively. We measure the time efficiency by conducting our experiments on a 2.6 GHz Intel Core i5 with 4GB of memory. As expected, the less expensive technique is Tarantula, while CA-MFL needs slightly more time.

With respect to space complexity, Ochiai needs two store the counters $n_{11},n_{10},n_{01},n_{00}$ for the similarity computation for all M components, and M represents the size of the object program. Hence, the space complexity is $O(M)$. CA-MFL also stores similar counters, and it has $O(M)$ space complexity.

5. Conclusions

In this paper we have presented a multiple-fault localization technique, coined CA-MFL. Our synthetic experiments using a benchmark set of well-known programs have confirmed that CA-MFL consistently outperforms other renowned techniques such as Ochiai, Jaccard and Tarantula. Although our evaluation clearly demonstrates a trade-off between reduction and fault-localization effectiveness, faults are due to the choice of test samples and the test suites have a highly biased sample of passing and failing test cases. Specifically, we plan to study different kinds of programs containing more faults that may provide greater ability to generalize the results. We also would like to explore adaptive strategies that enable the way of examining code adapt to the structure of code to provide more alternatives in the trade-off between reduction and fault-localization effectiveness.

Acknowledgements

This research was supported in part by the National Natural Science Foundation of China (No.61379036, 61562015, 61502430); Major scientific and technological special key industrial projects of Zhejiang Province, China (Grant No. 2014C01047); the Natural Science Foundation of Zhejiang Province, China (No.LY12F02041); the Natural Science Foundation of Guangxi Province, China (No. 2015GXNSFDA139038); 521 ta-lent project of Zhejiang Sci-Tech University.

References

[1] H. Cleve and A. Zeller, "Locating causes of program failures." Proceedings of the 27th international conference on Software engineering. ACM, **(2005)**.

[2] A. Zeller, "Isolating cause-effect chains from computer programs." Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering. ACM, **(2002)**.

[3] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input." IEEE Transactions. Software Engineering, vol. 28, no. 2, pp. 183–200, February. **(2002)**.

[4] M. Weiser, "Programmers use slices when debugging." Communications of the ACM 25.7 **(1982)**: 446-452.

[5] H. Agrawal, R. A. DeMillo and E. H. Spafford, "Debugging with dynamic slicing and backtracking." Software: Practice and Experience 23.6 **(1993)**: 589-616.

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique." Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, **(2005)**.

[7] R. Abreu, P. Zoeteweij and A. J. C. Van Gemund, "On the accuracy of spectrum-based fault localization." Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, **(2007)**.

[8] R. Abreu, P. Zoeteweij and A. J. C. Van Gemund, "An evaluation of similarity coefficients for software fault localization." Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on. IEEE, **(2006)**.

[9] R. Abreu, A. Gonzalez-Sanchez and A. J. C. Van Gemund, "Exploiting count spectra for bayesian fault localization." Proceedings of the 6th International Conference on Predictive Models in Software Engineering. ACM, **(2010)**.

[10] S. Artzi, J. Dolby, F. Tip and M. Pistoia, "Fault localization for dynamic Web applications." Software Engineering, IEEE Transactions on 38.2 **(2012)**: 314-335.

[11] W. Wen, B. Li, X. Sun and C. Liu, "Technique of software fault localization based on hierarchical slicing spectrum." Journal of Software 24.5 **(2013)**: 977-992.

[12] L. Zhao, L. Wang, D. Gao, Z. Zhang and Z. Xiong, "Mining associations to improve the effectiveness of fault localization." Jisuanji Xuebao (Chinese Journal of Computers) 35.12 **(2012)**: 2528-2540.

[13] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact." Empirical Software Engineering 10.4**(2005)**:405-435.

[14] M. J. Harrold, G. Rothermel, R. Wu and L. Yi, "An empirical investigation of program spectra." ACM SIGPLAN Notices. Vol. 33. No. 7. ACM, **(1998)**.

[15] D. Zhang, J. Jiang and L. Chen, "A method for validating the effectiveness of fault clustering and failure clustering of programs." SCIENTIA SINICA Information is 10 **(2014)**: 011.

[16] C. Liu, X. Zhang and J. Han, "A systematic study of failure proximity." Software Engineering, IEEE Transactions on 34.6 **(2008)**: 826-843.

[17] M. Levandowsky and D. Winter, "Distance between sets." Nature 234.5323 **(1971)**: 34-35.

Authors

Xiao'an Bao, born in 1973, M.S. He is currently a full Professor in the institute of software, Zhejiang Sci-Tech University, China. He mainly researches adaptive software, software testing and intelligent information processing.

Yusen Wang, born in 1990, Master candidate. Her main research interests include fault localization technology.

Junyan Qian, born in 1973, He holds a Master degree in Computer Science from Guilin University of Electronic Technology, China, and a Ph. D. degree in Software Engineering from Southeast University, China. His research interests include formal software verification, model checking and real-time system.

Zijian Xiong, born in 1991, Master candidate. His main research interests include software testing technology.

Na Zhang, born in 1977. She holds a Master degree in Computer Application from Zhejiang University, China. She is an associate Professor in the institute of software, Zhejiang Sci-Tech University, China. She mainly researches software engineering and software testing technology.

Chenghai Yu, born in 1975,M.S. He is an associate Professor in the institute of software, Zhejiang Sci-Tech University, China. He mainly researches software engineering and software testing technology.