Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 12, December 2019, pp. 3179-3186
DOI: 10.23940/ijpe.19.12.p9.31793186

Search-based Software Debugging using Weighted Fault Propagation Graphs

Bingwu Fang[a,b], Yong Li[b,c,*], Yong Wang[b,d,e,*], Xiangyu Cheng[e], and Zhaohui Xu[f]

[a]*School of Yungui Information, Anhui Finance and Trade Vocational College, Hefei, 230601, China*
[b]*Key Laboratory of Safety-Critical Software (Nanjing University of Aeronautics and Astronautics), Ministry of Industry and Information Technology, Nanjing, 210000, China*
[c]*Data Security Key Laboratory, Xinjiang Normal University, Urumqi, 830054, China*
[d]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210000, China*
[e]*School of Computer and Information, Anhui Polytechnic University, Wuhu, 241000, China,*
[f]*Avic East Optoelectronics (Shanghai) Co., LTD, Shanghai, 201100, China*

Abstract

Manual program debugging is tedious as well as time-consuming. The high costs have motivated the development of automatic program debugging approaches, which mainly focus on helping programmers identify fault locations. Xie et al. revisited automatic debugging via human focus-tracking to validate its effectiveness. However, their observations implied that there exists interference between the mechanism of the automated debugging approach and the actual assistance needed by programmers during program debugging. To solve this problem, we propose a search-based software debugging approach based on weighted fault propagation graphs (WFPG). We firstly use spectrum-based fault localization techniques to generate a suspicious module-fault ranking list and then construct a WFPG for each suspicious program module to assist programmers in understanding fault propagation. Our approach integrates automatic fault localization and program understanding to help programmers debug. We conduct a case study to demonstrate the effectiveness of our approach, and the results are promising.

Keywords: debugging; fault localization; fault propagation graph

(Submitted on September 15, 2019; Revised on October 16, 2019; Accepted on November 25, 2019)

1. Introduction

Software debugging is a process of fault localization and fault correction after faults have been detected by testing in a program [1-2]. In the procedure of fault localization, programmers develop hypotheses about the fault locations and their root causes, and then they verify or refute these hypotheses by examining the program. The programmers again develop hypotheses about how to modify the program and then verify or refute them in the procedure of correction. Manual program debugging is tedious as well as time-consuming. The high costs have led to research on automated debugging approaches, which have been widely studied over the past decades [3-5]. In this community, researchers have been mainly focused on proposing new approaches and evaluating their effectiveness without considering fault understanding [6]. The results outputted by current automated debugging approaches are only approximate, and their correctness is generally not guaranteed. Therefore, programmers must understand and verify the results in processing debugging.

Parnin and Orso firstly conducted an empirical study to compare the actual help for programmers in debugging with and without automatic debugging techniques [7]. As a preliminary study, their work explicitly indicated that it is important to understand program bugs, as this determines the helpfulness of automatic debugging techniques. Xie et al. reconfirmed the importance of program comprehension and highlighted that the automated debugging technique may interfere with fault comprehension and fault localization [8]. To solve the issue, some techniques were proposed to help programmers understand the root cause of failures. Deng and Jones presented a weighted, hybrid system dependence graph model

* Corresponding author.
E-mail address: yongwang@ahpu.edu.cn, liyong@live.com

(*WSDG*), which represents the relevance of highly related, dependence code, to assist fault comprehension for multiple software-engineering tasks [9]. Cheng et al. proposed a bug signature mining technique from the correct and faulty execution of a program, which provides the context in which the failure occurs, and it has been used to help programmers understand the root faults of failures [10]. Perez and Abreu proposed a spectrum-based feature comprehension (*SFC*) that borrows techniques used for automated software fault localization, and it was proven to be effective even for large resource-constrained applications [11].

As a further step along this direction, we propose a search-based software debugging based on weighted fault propagation graphs to further improve the effectiveness of automatic program debugging. We firstly use spectrum-based fault localization techniques to generate suspiciousness module-level fault ranking and then construct a *WFPG* based on program execution traces for each hypothesis program module to help programmers understand the fault. Our approach integrates fault localization and program understanding to help programmers debug the buggy program. Our approach starts a program debugging task using a search-based strategy, which conforms naturally with manual program debugging. Our *WFPG* is different from the weighted system dependency graph model *WSDG* [9]. *WSDG* includes static dependencies as well as information about any number of executions, which inform the weight and relevance of the dependencies. *WSDG* represents the relevance of highly related, dependence code to assist developer comprehension. In contrast, *WFPG* combines dynamic control flow with weighted information about fault propagation, which is more suitable for fault understanding.

The rest of the paper is organized as follows. Section 2 gives an overview of program debugging. Section 3 presents our approach. A case study to validate the effectiveness of our approach is provided in Section 4, and Section 5 concludes the paper.

2. Overview of Debugging

Program debugging is a process of "diagnosing the root cause of a known fault and then correcting it". Debugging includes two tasks: *fault localization* and *fault correction*. When programmers detect that the program contains a fault(s), they make assumptions about the fault and its root cause and verify or refute these assumptions by checking the program. In the process of fault correction, the programmers again raise some assumptions about how to fix the fault and then verify or refute these assumptions. Generally, the program debugging process model seeks to validate a set of hypotheses, modify the set of hypotheses, select hypotheses for validation, and validate or refute the selected hypotheses until the fault is corrected.

Figure 1 shows a debugging framework. In the framework, the fault localization and fault correction processes are divided into a single loop of hypothesis and verification. The debugging process model begins with a fault report as the initial set of hypotheses. There are many fault localization techniques, such as spectrum-based fault localization techniques [12], which can generate a suspicious fault ranking report. During the debugging process, programmers will modify the hypothetical set by generating, refining, and checking. Randomly selecting a hypothesis to verify is usually not valid. Instead, a hypothetical fault that is ranked highly in the fault reports is usually chosen. Programmers use certain techniques to verify the hypothetical fault by examining the program and its dynamic behaviors during its execution. Using these results, they can prove that the suspicious fault is correct or wrong.

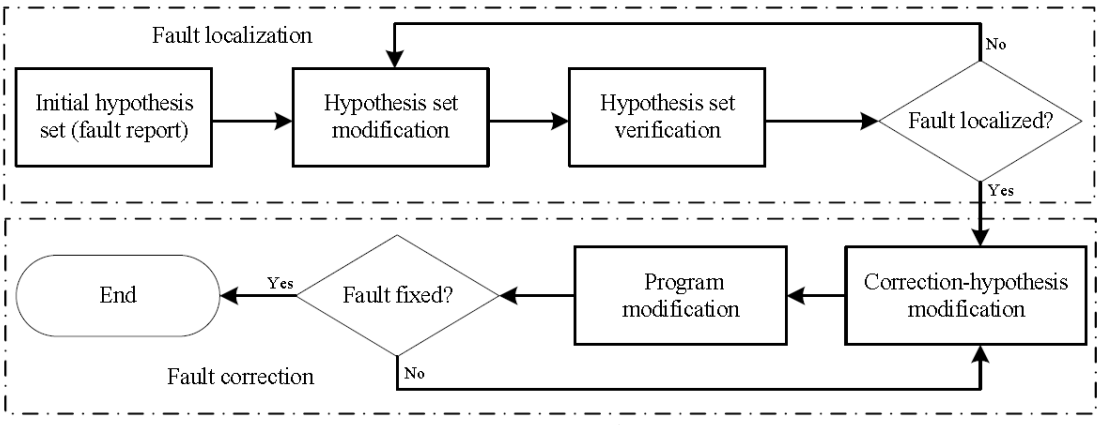

Figure 1. A debugging framework

Fault localization does not exist independently in program debugging. In reality, to localize the root causes of failures, programmers need to understand the program fault and how to cause and propagate program failures in failed testing. Therefore, it is easy to fix a program's fault when it is localized. Fault understanding is the key to overall program debugging. In our approach, we construct a *WFPG* to help programmers understand program fault propagation.

3. Our Approach

3.1. Framework

Let $P = \{m_1, m_2, \cdots, m_k\}$ be a buggy program, and P contains k modules. $T = \{t_1, t_2, \cdots, t_n\}$ is a set of test cases. Similar to other spectrum-based fault localization techniques, we first collect program spectra, then measure the program module's suspiciousness, and generate a module ranking list of P in descending order based on the suspicious scores. Finally, we choose a hypothesis module from the ranking list to check. During this time, we construct a *WFPG* for the hypothesis fault module to further support block-level fault localization and understanding. The framework is shown in Figure 2. The automatic program debugging framework mainly includes two steps: suspicious module-fault localization and fault understanding. In our approach, we use a spectrum-based fault localization technique to perform suspicious module-fault localization and construct a *WFPG* to help programmers understand fault propagation.

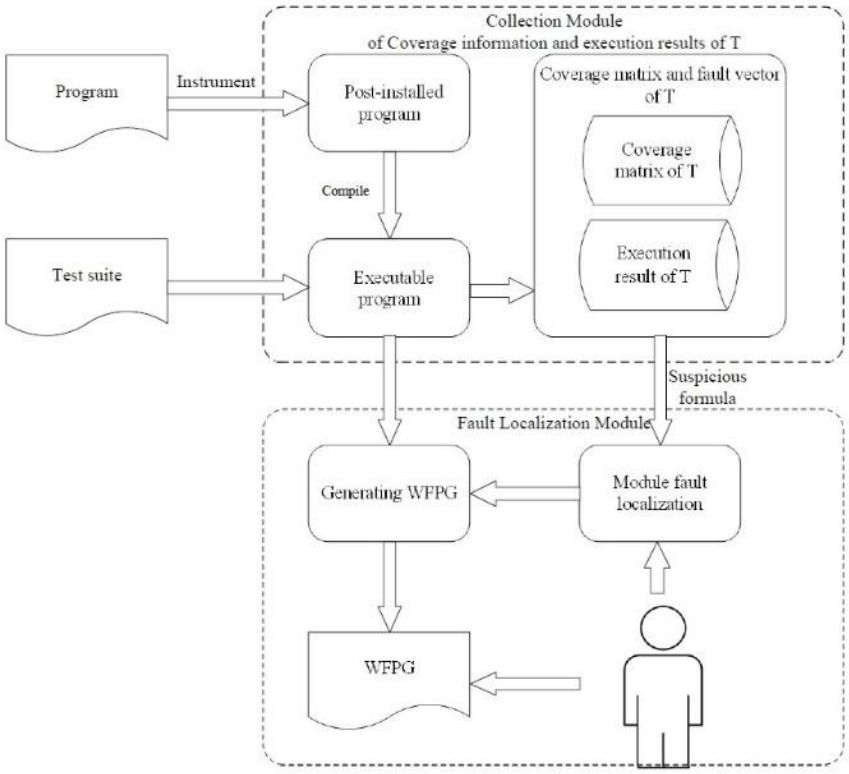

Figure 2. Search-based program debugging framework

3.2. Spectrum-based Fault Localization

Spectrum-based fault localization techniques calculate the likelihood of a program component being the root cause of failures for each component in a program [3]. Suppose there is a program $p = \{s_1, s_2, \cdots, s_m\}$ and a set of test cases $T = \{t_1, t_2, \cdots, t_n\}$ for the program including m test cases, where T can be divided into T_p and T_f according to the success or failure of testing results. Then, the program spectrum can be expressed as a two-dimensional $n \times m$ matrix M, where

$$M_{ij} = \begin{cases} 1, \ T_i \text{ covered the program component } s_j \\ 0, \ \text{Otherwise} \end{cases} \quad (1)$$

For a test suite $T = \{t_1, \cdots, t_2\}$, $t_i = (ipt_i;\ o_i)$, where ipt_i is an input value for testing t_i, o_i is an expected output value for test execution. The testing results of T are represented as $e = \{e_1, e_2, \cdots, e_n\}$, where

$$e_i = \begin{cases} 1, \text{ if } (ipt_i\ !=o_i) \\ 0, \text{ Otherwose} \end{cases} \tag{2}$$

The pair (M, e) is inputted for a spectrum-based fault localization approach. The program spectrum M and result vector e are shown in Figure 3.

In general, spectrum-based fault localization techniques mainly include the following three mainly steps: (1) test the program and collect the pair (M, e). (2) Select the corresponding suspicious measurement formula to calculate the suspicious score for each program component. The higher the suspicious score of the program component, the higher the possibility that it contains faults. (3) Rank the suspiciousness of each program component. Each program component is sorted in descending order of suspicious score.

$$\begin{array}{c} \\ N \text{ spectra} \end{array} \begin{array}{c} \overset{M \text{ components}}{\overbrace{\phantom{\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{M2} & \cdots & a_{NM} \end{bmatrix}}}} \\ \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1M} \\ a_{21} & a_{22} & \cdots & a_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ a_{N1} & a_{M2} & \cdots & a_{NM} \end{bmatrix} \end{array} \begin{array}{c} \text{error} \\ \text{detection} \\ \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \end{array}$$

Figure 3. Program spectrum and error detection vector

Given (M, e) and the suspiciousness metric ρ, the result of fault localization can be determined. The module-fault localization algorithm is described in Algorithm 1. We define the spectrum-based fault localization as map function *Map*, where

$$Map(M, e, \rho) \rightarrow faultRanking \tag{3}$$

3.3. Weighted Fault Propagation Graph

We first give a formal definition of control flow graph (*CFG*). The *CFG* is a static program model, and it is used to determine the control flow for statements/blocks in a program that is executed subsequently.

Each component c_i in program $P = \{c_1, \cdots, c_n\}$ is represented as a directed graph G_{cfg} = (V, E, e, x), with nodes V, edges E, entry vertex e, and $exit$ node x. In a *CFG*, V represents the node set, and the nodes correspond to the program components, which can be statements, predicates, methods, modules, or classes in a program. E is the set of edges, and it represents the control relationship between nodes. e and x are two special nodes in N. Other nodes in N can start from the *entry* node and arrive at the *entry* node in the program execution. Similarly, any node in N has a path ending at the *exit* node. Node e has no input edge, and node x has no output edge.

Algorithm 1 Spectrum-based Module-fault Localization Algorithm

Input: Program P, set of test cases T, and metric ρ_m;
Output: Fault Report D;

```
N ← |T|;
M ← getNumberofModules(P);
D ← ∅;
for i = 0 to M do
  {n00(i), n01(i), n10(i), n11(i)} ← 0;
  Susp[i] ← 0
end for
(M, e) ← RunProgram(P, T);
for i = 0 to M do
  for j = 0 to N do
    if M[i, j] = 1 ∧ e[j] = 1 then
      A11(j) ← A11(j) + 1
    else if M[i, j] = 1 ∧ e[j] = 0 then
      A10(j) ← A10(j) + 1
    else if M[i, j] = 0 ∧ e[j] = 1 then
```

```
16:          A_01(j) ← A_01(j) + 1
17:       else
18:          A_00(j) ← A_00(j) + 1
19:       end if
20:    end for
21: end for
22: for i = 0 to M do
23:    Susp[i] ←ρ_m (A_00(j), n_01(j), n_10(j), n_11(j));
24: end for
25: normalize(Susp);
26: D ← sort(Susp);
27: return D;
```

According to the result of program executions, there are two sets of faulty and correct execution traces. In the failed trace set, the faults are triggered and the error states are propagated, resulting in program failures. In the debugging activity, we focus only on localizing faults, understanding them, and correcting them. Inspired by [13], when a failure is observed during program execution, the fault location should be reached, and the inter-state should be infected and propagated with control flow until it is observed. Therefore, we are interested in constructing a fault propagation graph (*WFPG*) using faulty execution traces to help programmers understand faults.

A block-level fault propagation graph $FPG\ G_b = (N, E)$ for program P is a directed graph, which consists of a set of nodes N and a set of directed edges E, where N represents the program's blocks $b_1, \cdots, b_n$, and E is a set of edges that show how the fault states can be propagated from block to block. There exist three types of edges:

- An edge $\langle v_i, v_j \rangle \in E(G_b)$ is labeled as *trans* if and only if block j is executed right after block i in an execution trace tr_b by the failed test case.
- An edge $\langle v_i, v_j \rangle \in E(G_b)$ is labeled as *call* if and only if block j is called by block i in an execution trace tr_b by the failed test case.
- An edge $\langle v_i, v_j \rangle \in E(G_b)$ is labeled as *return* if and only if it corresponds to the method *return*, where block i returns to block j in an execution trace tr_b by the failed test case.

A block-level fault propagation graph G_b is difficult to understand due to its complex control flow. Therefore, we only consider its subgraph for a hypothesis fault module m_i . Giving a hypothesis module m_i , a *WFPG* is defined as $WG_b(N, E, W_N, W_E, M_i)$, where N and E are sets of nodes and edges similar to *FPG*, W_N and W_E are weighted information about N and E, respectively, and m_i is a hypothesis module that programmers want to verify.

There are several similarity and distance measurements that can be used [14]. We utilize *Ochiai*, which a similarity metric that can assess the correlation of a node to program failures. The weight of nodes N_i is defined in Equation (4).

$$W_N^i = \frac{n_{11}(N_i)}{\sqrt{totalfailed(N_i) \times (n_{11}(N_i) + n_{10}(N_i))}} \tag{4}$$

We use the *Jaccard* metric to compute the weighted information for E_i. For $E_i = \langle n_i, n_j \rangle \in E$, W_E^i is defined as follows:

$$W_N^i = \frac{n_{11}(E_i)}{n_{01}(E_i) + n_{10}(E_i) + n_{11}(E_i)} \tag{5}$$

3.4. Visualization

In the process of fault localization based on *WFPG*, the programmers first perform the module-level fault localization. Then, fine-grained fault localization is implemented to build a *WFPG* for the modules that must be further checked. Therefore, it is necessary to visualize a *WFPG* for a module.

We use *Networkx* tools (see http://networkx.github.io in detail) to construct the *WFPG* and visualize it. *Networkx* is a Python toolkit for complex network applications. Generally, building a graph model is about determining the corresponding sets of nodes and edges. The edge set and node set of the *WFPG* can be easily obtained. Construct a *WFPG* that identifies the main *entry* and *exit* nodes and processes the function calls in the graph. If there is no uniform *exit* sink, a "dummy node"

is added as the sink node for the convenience of understanding. Therefore, it is easy to build a *CFG* through the program execution traces collected during failed test executions. To facilitate program understanding, the execution trace collected during the execution of successful test cases is used to complete the *CFG*, which is represented by a dotted line. This part represents that the nodes have not been executed by failed testing.

We use different colors and node sizes to visualize the node weight information. Based on the weighted information of nodes, the nodes in different fault rankings are filled with different colors. Consider the large difference between weighted information of nodes in a *WFPG*. For example, in the node set, the minimum weight is *0.01* and the maximum weight is *6*, with a difference of *600* times. The direct use of weights to control node size is inappropriate. To solve this problem, we map the node size according to the ranking of the node weight so that it can reflect the node weight information through the node size.

4. Case Study

In this section, a case study is used to evaluate the effectiveness of *WFPG* and help programmers understand the faults. We choose a program *replace* as our study case. The *replace* is a pattern matcher that takes three inputs, including the string *s*, the pattern *p* to be identified in *s*, and the string *r*. Figure 4 shows a snippet of the module *esc* that has been inserted into version 20 of *replace* to identify and process *escape* characters in *p* and *s*. The program contains a bug on line 89, and the correct statement should be *result = ESCAPE*. However, during software development, a programmer accidentally wrote *result = ENDSTR*. During program testing, 220 test cases failed to execute in 5,542 test cases, and the other test cases executed successfully. The program *replace* contains 21 modules and 512 lines of code. *Esc* is the module containing bugs and contains a total of nine basic blocks. To facilitate program understanding, a sink node 10 is added to this module. During the program execution, the block-level execution trace is saved into the profile file with the instrumentation code.

lines	blocks	source code
75		char esc(s, int i)
76		char *s;
77		int *i;
78		{ char result;
79		fputs("esc invoke\n",proonfile);
80	1	fputs("esc-1\n",profile);
81		if(s[*i] != ESCAPE)
82		{
83	2	fputs("esc-2\n",profile);
84		result = s[*i];}
85		else
86	3	{ fputs("esc-3\n",profile);
87		if(s[*i + 1] == ENDSTR)
88	4	{ fputs("esc-4\n",profile);
89		result = ENDSTR; /* bug*/
.....	.....	.....

Figure 4. Esc program segment

According to the software debugging framework described in the previous section, the module-level fault localization is performed first, and then a *WPPG* is generated for the suspiciousness module. The left side of Figure 5 shows the module call graph generated for the program *replace*. In the program debugging framework, the suspicious module is firstly localized, and then the basic block-level fault localization is further performed based on *WFPG*. In this case, the suspicious fault module *esc* is firstly pinpointed, and then a *WFPG* is constructed for the module *esc*. The corresponding *WFPG* is shown on the right in Figure 5. The *WFPG* shows that the root cause of software failure is the basic block 4 (*esc-4*). Base block 3 (*esc-2*) is the fault propagation context for base block 4 (*esc-4*); therefore, base block 3 also has a higher suspiciousness score than other basic blocks. Therefore, it is easy for programmers to understand fault propagation and localize through search-based program debugging. With search-based program debugging, it is easier for programmers to understand program faults during debugging.

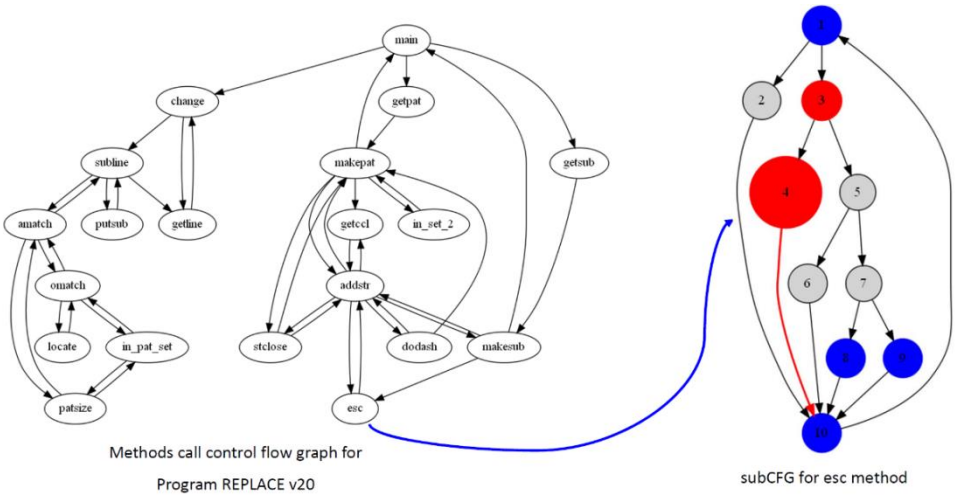

Figure 5. A case study

5. Conclusions

Aiming at the context of fault propagation, we propose a search-based software debugging based on weighted fault propagation graphs. The approach focuses on helping programmers understand program failures during debugging. In the approach, we first introduce an overview of program debugging. Then, the search-based program debugging framework is proposed, and spectrum-based fault localization techniques and $WFPG$ are introduced. Finally, we validate the effectiveness of our approach through a case study.

In the future, there are two main directions to continue this research work. First, it is proposed to establish an algorithm to predict the effectiveness of module-level fault localization based on program spectrum. Second, it is planned to build a fault execution environment with the help of *WFPG* to further help programmers understand programs' faults.

Acknowledgments

This work was supported by the Anhui Natural Science Foundation (No. 1908085MF183, 1608085MF147), Anhui University Natural Science Fund Key Project (No. KJ2017A859, KJ2018A0116, KJ2016A252, KJ2017A104), National Key Research and Development Program (No. 2016YFB100), NSFC of China (No. 61976005, 61772270, 61562087), State Key Laboratory Research Program for Novel Software Technology (Nanjing University) (No. KFKT2019B23), and Safety-Critical Software Key Laboratory Research Program (No. NJ2018014).

References

1. K. Araki, Z. Furukawa, and J. Cheng, "A General Framework for Debugging," *IEEE Software*, Vol. 8, No. 3, pp. 14-20, 1991
2. A. P. Mathur, "Foundations of Software Testing," Pearson Education India, 2013
3. W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A Survey on Software Fault Localization," *IEEE Transactions on Software Engineering*, Vol. 42, No. 8, pp. 707-740, 2016
4. Y. Wang, Z. Huang, Y. Li, and B. Fang, "Lightweight Fault Localization Combined with Fault Context to Improve Fault Absolute Rank," *Science China Information Sciences*, Vol. 60, No. 9, pp. 092113, 2017
5. Y. Wang, Z. Huang, B. Fang, and Y. Li, "Spectrum-based Fault Localization via Enlarging Non-Fault Region to Improve Fault Absolute Ranking," *IEEE Access*, Vol. 6, pp. 8925-8933, 2018
6. S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, et al., "Evaluating and Improving Fault Localization," in *Proceedings of the 39th International Conference on Software Engineering*, pp. 609-620, Buenos Aires, Argentina, May 2017
7. C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pp. 199-209, Toronto, Canada, July 2011
8. X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu, "Revisit of Automatic Debugging via Human Focus-Tracking Analysis," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 808-819, Austin, USA, May 2016
9. F. Deng and J. A. Jones, "Weighted System Dependence Graph," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 380-389, Montreal, Canada, April 2012
10. H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures using Discriminative Graph Mining," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pp. 141-152, Chicago, USA, July 2009
11. A. Perez and R. Abreu, "A Diagnosis-based Approach to Software Comprehension," in *Proceedings of the 22nd International*

Conference on Program Comprehension, pp. 37-47, Hyderabad, India, June 2014
12. W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar Method for Effective Software Fault Localization," *IEEE Transactions on Reliability*, Vol. 63, No. 1, pp. 290-308, 2013
13. N. Li and J. Offutt, "Test Oracle Strategies for Model-based Testing," *IEEE Transactions on Software Engineering*, Vol. 43, No. 4, pp. 372-395, 2016
14. S. S. Choi, S. H. Cha, and C. C. Tappert, "A Survey of Binary Similarity and Distance Measures," *Journal of Systemics, Cybernetics and Informatics*, Vol. 8, No. 1, pp. 43-48, 2010