Software Fault Localization Based On Centrality Measures*

Ling-Zan Zhu, Bei-Bei Yin, and Kai-Yuan Cai
Department of Automatic Control
Beijing University of Aeronautics and Astronautics
Beijing, 100191, China
E-mail: zhulingzan@gmail.com

***Abstract*—The existence of software faults is the cause of software failure. Locating the faults hidden behind software is a central task towards efficient software development and maintenance. Many techniques have been developed to help locate software faults. However, they have some obvious shortcomings. Inspired by the surprisingly discovery in the crossover studies of software and complex networks, in this paper, a fault localization method based on software network centrality measures (SNCM) is proposed. Software execution traces are modeled as networks and two centrality measures (node degree and structural hole) are adopted to calculate the suspiciousness of each statement. Modeling software from a network view can help handle the increasing complexity involved in fault localization. Moreover, the software network measures are essential because they focus on the macro-level software statistics. The universality of the proposed method is thus promising. Statistically repeatable experiment is applied to the programs in the Siemens suite and the results demonstrate the effectiveness of the proposed method and the poor performance of the Tarantula method under random inputs.**

Keywords-Fault localization; software networks; complex networks; centrality measures

I. INTRODUCTION

As the wide use of software product, the quality of software has become an important issue. People hope that software applications have high reliability and high availability. The triggered software faults can lead to software failure, so software fault localization is a key step of software development and maintenance. The effectiveness of the fault localization directly influences the software availability[1].

The existence of bugs in a program is often observed when the program output deviates from the expected output. A standard debugging process consists of setting breakpoints, re-executing the program on the failed input, and examining the program states to understand the cause of the failure. However, there are often too many states to be checked and the process of locating the faults is often tedious and time consuming. Therefore from the early 1970's, different methods have been proposed to help the programmers locate the faults. These methods can be roughly classified into the following four types: state-based methods[2-4], slicing-based methods[5-7], spectrum-based methods[8-10], and statistical model-based methods[11-12]. Section II will further introduce the related studies.

*Supported by the Open Project Program of the State Key Laboratory of Software Development Environment, and the National Science Foundation of China under Grant No. 60973006

Although the existing methods have been reported to be effective by conducting some case studies, they are still weak in several aspects:

(1) There exists no universal fault localization method. The complexity of software makes the nature of software obscure and lack of being well understood. It is not clear how to describe the software behavior, especially the software dynamic behavior, to a quantitatively high precision. Therefore, the proposed methods are most concerned with certain software properties rather than the universal ones, which make the methods hard to be universal.

(2) Besides the coverage information, more essential dynamic information which can reflect the software structure or properties should be used to help localization. Although coverage is important, more essential information related to software failure behavior may exist and should be extracted to help fault localization.

(3) The repeatability of experiments conducted by the reported methods is questionable[13]. Toy subject processes or products rather than real applications are often employed. Besides, the collected data may not be statistically repeatable.

In this paper, a fault localization method based on software network centrality measures (SNCM) is proposed, as a possible solution to the problems in the existing methods we concluded before. A software network treats software systems as a complex network, with software units (e.g. statements, functions, classes, objects) as nodes, and the static or dynamic relations (e.g. the reference between two objects) as edges. A complex network usually refers to a large-scale network that is subject to various forms of uncertainty and evolution. Due to its emphasis on finding general rules instead of analyzing details, complex network is treated as a powerful tool to help handle problems of system complexity in recent years. Besides, complex network stems from graph theory and statistical physics. It has a sound foundation of mathematics which is helpful to find quantitative rules underlying the complex systems. In recent years, some typical measures in complex networks (e.g. degree, average path length, clustering coefficient, betweenness) are adopted in the research of software networks. And several quantitative rules underlying software static structure[14-16] and dynamic behavior[17] are revealed based on these measures. To sum up, complex network is a suitable model to describe the complex software static structure and dynamic behavior. The measures obtained from the software networks can reflect some essential software properties.

978-0-7695-4459-5/11 $26.00 © 2011 IEEE
DOI 10.1109/COMPSACW.2011.17

In this paper, software statements are treated as nodes, and an execution of statement i followed by an execution of statement j defines an edge from node i to j. Thus the execution traces can be modeled as undirected networks. Two measures related to node centrality are adopted to evaluate the suspiciousness of the statements. One is the node degree, and another is structural hole. The degree of a node measures how many edges connected to it, while the structural hole of a node measures the "bridge" ability of the node to connect non-duplicated nodes. The method is proposed based on the knowledge that the faulty statement may play different role in the network constructed by the execution traces in the failed trials and the network constructed by those in the successful ones.

As indicated before, an important weakness of the existing methods is that the experimentation paradigm adopted is questionable. For one thing, the employed subject program should be chosen carefully to make the experiment realistically. The size of the subject program should be reasonably large and more than two subject programs should be employed to valid the repeatability of the results. However, the existing methods often use one program of small size to demonstrate their effectiveness. For example, the widely used Siemens test suite contains 7 programs, among which the largest one *print_tokens* only have 539 lines of code. For another, the collected data may not be statistically repeatable. The existing methods, Sober and Tarantula for example, run the whole test case suite and use the information to locate faults. However, in practice, we run the software program according to a certain profile instead of running all possible inputs exhaustively. The experimental results thus may not be valid under random inputs, i.e. they may not be statistically repeatable. Different from the existing methods, the experimentation paradigm with statistical repeatability was applied to the programs in the Siemens suite and the results demonstrate the effectiveness of the proposed method. The program was run many rounds, the number of which is large enough to have a statistical sense. In each round, test cases selected from the test suite were input one by one randomly according to a certain testing profile. The final result was obtained by averaging the results of the rounds. Besides, we applied the Tarantula method using both the non-repeatable experimentation paradigm and the repeatable one to show its poor performance under random input, which emphasizes the importance of adopting the experimentation paradigm with statistical repeatability and the need to re-study the effectiveness of the existing methods.

The reminder of this paper is organized as follows. Section II introduces the related studies of software fault localization. Section III explains the proposed SNCM method. Section IV reports the case study and the results. Conclusion and future work are included in Section V.

II. Related Studies

Different methods have been proposed for fault localization, including[18]:

(1) State-based methods. Such as the delta debugging methods[2], the cause transition method[3], the predicate switching method[4], etc. Similar to the traditional debugging methods, these methods still focus on the searching of the cause of incorrect output being generated. However, by using these methods, the amount of the states that need to be checked can be reduces a lot.

(2) Slicing-based methods. Such as the static slicing-based methods[5], the dynamic slicing-based methods[6], and the execution slicing-based methods[7], etc. The static slicing-based methods find the set of all statements that might affect the value of a variable to locate the faults, while the dynamic slicing based methods find all statements the really affect the value of a variable for the given program inputs. An execution slicing is the set of codes executed by a given test case. The execution slicing-based methods often give the suspicious statements by finding the differences between the execution slices of the successful tests and the failed tests.

(3) Spectrum-based methods. Such as the Tarantula method[8] and Ochiai[9], etc. M. J. Harrold[10] proposed several program spectra which are often used in program debugging (e.g. Branch Hit Spectra, Executable Statement Hit Spectra, Complete Path Spectra, etc.). These two methods use the Executable Statement Hit Spectra, which describes the coverage of executable statements, to locate the faults.

(4) Statistical model-based methods. Such as the Sober method[11], the Liblit method[5], and the Cross-tab method[12], etc. These methods often instrument the executable statements or predicates and record the coverage of them during runtime. By analyzing the coverage information of the successful tests and the failed tests, these methods calculate the suspiciousness of each executable statement or predicate and examine them according to the suspiciousness ranking from high to low to locate the faults.

We further introduce the Tarantula[8] method briefly here, as it is used to compare with the proposed method in the case study presented later. The performance of Tarantula is shown to be better than other fault localization methods (such as set-union, set intersection, nearest-neighbor, and cause-transitions with respect to the Siemens suite). Thus we only focus on the comparison between the effectiveness of the Tarantula method and that of our method. The Tarantula method uses the coverage information collected at runtime, and the execution results of each trial being successful or failed to calculate the suspiciousness of each statement as:

$$suspiciousness(s) = \frac{N_{CF}(s)/N_F}{N_{CF}(s)/N_F + N_{CS}(s)/N_s}$$

where $N_{CF}(s)$ and $N_{CS}(s)$ are the number of failed and successful tests that execute the statements respectively. N_F and N_s are the total number of failed and successful tests respectively.

III. The Proposed Technique

A. Centrality Measures

Within graph theory and network analysis, there are various measures of the centrality of a node within a

network that determine the relative importance of a node within the network. For example, how important a person is within a social network, or how well-used a road is within an urban network. Typical centrality measures include degree centrality, betweenness centrality, closeness centrality, eigenvector centrality, etc. In this paper, we adopt two centrality measures into the proposed technique, one is degree centrality, and another is structural hole. Figure 1 illustrates the two measures.

Degree centrality is the simplest centrality measure and is defined as the number of edges that a node possesses. Degree is often interpreted in terms of the immediate risk of node for catching whatever is flowing through the network (such as a virus, or some information). In Figure 1, the degree of node F is the largest, which implies it may be a critical node in the network. For example, in epidemic dynamics, node F has more risk to be infected or spread the disease because it has the most connections with others. However, the immunization on the node works best for the same reason.

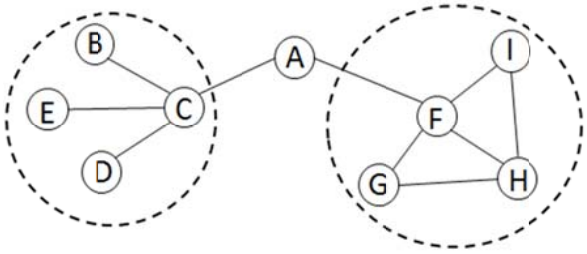


Figure 1. Example of Node Centrality

Next let us notice a special node A in the network whose degree centrality is low because it has only two connections. However, we can find that node A bridges two small networks which cannot connect to each other without it. Node A thus plays an important role in the network structure. Structural hole can be used to measure this "bridge" ability of nodes. The concept of structural hole is from the social structure of competition[19]. It describes the relationship between two non-duplicate persons. For example, in the left small network in Figure 1 which is formed by nodes B, C, D and E, there are three structural holes (BE, BD and DE) because they have no direct connection (non-duplicate). Node C here plays the bridge role and is associated with these three nodes. Compared with the other three nodes, node C is in the center and has competitive advantage. However, the right small network in Figure 1, which is formed by nodes F, G, H, and I, is a closed network and has no structural hole. Similarly, node C and node F are two non-duplicate nodes and has a structural hole between them, which is bridged by node A.

There are two ways to measure the structural hole. One is the Burt coefficient of a node proposed by Burt, and another is the betweenness of a node which defined as the number of shorted paths between two other nodes that occur on the node. We use the Burt coefficient in this paper to measure the structural hole. The Burt coefficient bt_i of node i equals to the ratio of its effective size to its real size (its node degree d_i):

$$bt_i = ES_i / d_i \tag{1}$$

where $1 \le i \le N$ and N is the number of statements in the program. The effective size ES_i of node i equals to the node degree d_i of the node minus the redundancy of the node. The redundancy r_i of node i is calculated by Equation (2):

$$r_i = \frac{\sum_j d_j^i}{d_i} \tag{2}$$

Suppose node i has n neighbors, that is, those nodes that connected directed to node i. Node i and its neighbors with the connections among them construct a small network called the neighbor network of node i. d_j^i in Equation (2) represents the degree of node j in the neighbor network of node i, where $1 \le j \le n$. Thus the effective size ES_i of node i is:

$$ES_i = d_i - r_i \tag{3}$$

B. Definition of Suspiciousness

In this paper, we propose a fault localization method based on software network centrality measures: degree and structural hole. In Part A of section III we introduced these two measures which can reflect the important role of nodes in a network from different aspects.

Run a program for a trial, we get the testing result: failed or successful. The execution trace is called failed trace or successful trace correspondingly. After some trials, a collection of failed traces Tr^F and a collection of successful traces Tr^S were obtained. We treat software statements as nodes, and an execution of statement i followed by an execution of statement j as an edge between node i and j. Thus the collection of failed traces and successful traces can be modeled to two networks G_F and G_S respectively. The SNCM method is proposed based on the knowledge that the faulty statement behaves differently in the failed trials and the successful trials, that is, the node of the faulty statement plays different role in the structure of G_F and G_S. The suspiciousness S_i of node i is defined as Equation (4):

$$S_i = \frac{d_i^F}{d_i^F + d_i^S} + \frac{bt_i^F}{bt_i^F + bt_i^S} \tag{4}$$

where d_i^F and bt_i^F are the degree and Burt coefficient of node i in the faulty trace network. And d_i^S and bt_i^S are those of node i in the successful trace network.

The first item of the right side in Equation (4) is not hard to be understood. The execution of the faulty statement is the precondition to trigger the fault. Thus the ratio of the faulty statement being executed in the faulty trials to the total times of it being executed tends to be larger than the other statements. As for the second item, we know that the Burt coefficient is the measure of the structural hole. The fault is triggered when the condition is satisfied after the execution of the statements executed before the faulty one. And the influence of the fault convey to the other statements executed after the faulty one. The faulty statement position at the center of information transmission of the two parts, the corresponding Burt coefficient might be large as a result.

C. Procedure of Fault Localization

The procedure of the proposed fault localization technique is summarized as follows. Here we note that in the procedure, $1 \le i \le N$ and N is the number of statements in the program.

Step1: Model the traces in sets Tr^S and Tr^F to be Networks G_S and G_F respectively, with software statements as nodes, and an execution of statement i followed by an execution of statement j as an edge between node i and j.

Step2: Calculate the degree d_i^S and d_i^F of node i.

Step3: Calculate the Burt coefficient bt_i^S and bt_i^F of node i.

Step4: Calculate the suspiciousness S_i of each node i according to Equation (3).

Step5: Rank the statements $s_1, s_2, \cdots s_N$ based on $S_1, S_2, \cdots S_N$ in descending order and examine the statements from top until the faults are located.

Step6: End the procedure.

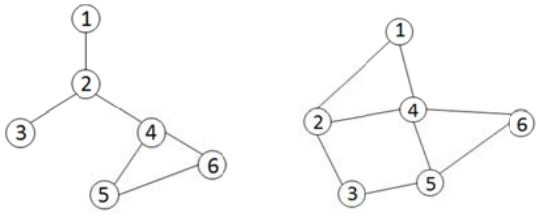
Let's further demonstrate our method through a simple example. We use the example presented in Table I. Suppose we have a program with 6 statements ($N = 6$) and one fault in statement s_2. We conducted five trials, two of which failed. In Table I, each row contains the execution trace and the execution result of a test case.

TABLE I. AN EXAMPLE OF THE EXECUTION TRACES

Test case	*Execution trace*	*Test result*
T1	1 2 4 5 6 4	Failed
T2	1 2 4 6 4 2 3	Failed
T3	1 2 3 5 4 6	Successful
T4	1 2 4 5 6 4 2 3 5	Successful
T5	1 4 5 6 4 2 3	Successful

The proposed method is applied as follows:

- Using the set $Tr^F = \{T1, T2\}$ to model the network G_F, and the set $Tr^S = \{T3, T4, T5\}$ to model the network G_S respectively. From execution trace T1, it is known that there is an edge between node 1 and node 2, node 2 and node 4, node 4 and node 5, node 5 and node 6, node 6 and node 4 respectively. Since it is not necessary to consider multiple edges between two nodes or directions of edges here, from T2, we can only get extra an edge between node 2 and node 3. Thus, the network G_F is generated, and shown in Figure 2(a). According to the same method, the network G_S is obtained from T3, T4, T5 and shown in Figure 2(b).



(a) Network G_F (b) Network G_S

Figure 2. G_F and G_S of the Example

- Calculate the degree d_i^S and d_i^F of node i. Calculate the Burt coefficient bt_i^S and bt_i^F of node i. Calculate the suspiciousness S_i of each node i according to Equation (3). The values are shown in Table II.
- Rank the statements $s_1, s_2, \cdots s_6$ based on $S_1, S_2, \cdots S_6$ in descending order and examine the statements from top until the faults are located. The localization result is shown in Table II.

TABLE II. THE LOCALIZATION RESULT OF THE EXAMPLE

Statement	d_i^F	d_i^S	bt_i^F	bt_i^S	S_i	Rank
1	1	2	1	0.5	1	2
2	3	3	1	0.778	1.062	1
3	1	2	1	1	0.833	5
4	3	4	0.778	0.75	0.938	4
5	2	3	0.5	0.778	0.791	6
6	2	2	0.5	0.5	1	3

The statements are examined one by one according to the ranking, and the fault will be found when statement 2 is examined. In this example, we examined 1 statement before we located the fault.

IV. CASE STUDY

A. Subject Programs

Siemens test suite is widely used as a benchmark for comparing different fault localization methods. The seven programs are also used in our study to show the effectiveness of the proposed method.

Siemens test suite contains 132 faulty versions of seven subject programs. Each faulty version contains only one fault. The information of the Siemens test suite is shown in Table III, including the number of faulty versions, LOC, the number of test cases, etc. Here we note that because the faults in two versions (the 4th and 6th versions of the program *print_tokens*) are in the header file instead of in the C file, these two versions are excluded in our study as well as in previous studies.

TABLE III. OVERVIEW OF THE SIEMENS SUITE.

Program	**No. of Faulty Versions**	**LOC**	**No. of Test Cases**	**No. of Stubbed Statements**
print_tokens	7	539	4130	93
print_tokens2	10	489	4115	205
replace	32	507	5542	75
schedule	9	397	2650	24
schedule2	10	299	2710	105
tcas	41	174	1608	16
tot_info	23	398	1052	73

B. Experimental Process

Different from the existing methods, the experimentation paradigm with statistical repeatability was applied to the programs in the Siemens suite. For each version of each program, 40 trials were conducted. In each trial, instead of running the whole test suite one by one, we selected 500 test cases in the sequence determined by the random strategy from the corresponding test suite. Here we note that the testing profile in this experiment is $\langle C_i, 1\rangle, i = 1,2,\ldots,num_{ts}$. num_{ts} is the number of test cases in the

corresponding test suite. Every test case in the test suite is treated as an equivalence class. The test strategy deciding which equivalence class to be selected next is the random strategy, which selects an equivalence class randomly.

For each version of each program in the Siemens test suite, one round of the experiment process can be described as follows:

Step1: Start the process.

Step2: Select a test case from the test suite at random.

Step3: Run the program with the selected test case. Record the execution traces and the results of the trial.

Step4: If test cases number doesn't reach 500, the process will go to Step2. Otherwise, go to Step 5.

Step5: Apply the SNCM method to locate the fault using the results of the 500 trials.

Step6: End the process.

After conducting 40 rounds of such process, we get 40 results of fault localization for each version. The average of the 40 results is the final localization result for the version. This is different from the existing methods which run the whole test suite one time for each version and take the result as the result of that version.

C. Experiment Results

We applied the proposed method, SNCM method, and the Tarantula method to the subject program according to the process described in Part B of section IV. The algorithm of the Tarantula method was introduced in Section II.

To quantify the localization accuracy, a measure called *T*-score is adopted in many previous studies, which estimates the percentage of code that has been examined in order to locate the fault.

$$T = \frac{|V_{examined}|}{|V|} \times 100\% \qquad (5)$$

There are mainly two evaluation frameworks called PDG-based and Ranking-based respectively to calculate the T-score. PDG-based framework, which is originally proposed by Renieris and Reiss[20], calculates the T-score based on static program dependence graph. Given a fault localization report R, which is a set of suspicious statements, the examiner starts from the statements in report R and perform a breadth-first search until one of the defect is located. Another Ranking-based framework, which is adopted by Tarantula[3], produces a ranking of all executable statements and examines them according to the ranking.

In this paper, the ranking-based method is adopted as it is in Tarantula. In Part B of section III, we reported the definition of suspiciousness S_i. The statements are ranked from high to low according to the suspiciousness and examined one by one until the fault is located. The T-score is then calculated by equation (5). Although in reference[5], it is claimed that the PDG-based method is closer to practice, the shortcoming of the method is also obvious. A program dependence graph (PDG), which contains the data and/or control dependencies between the statements, is needed before the fault localization method is applied. However, the documents of software are often inadequate due to poor development management. Not to mention the enormous open-source software, the documents of which are usually unavailable. The ranking-based method locates the faults totally depending on the dynamic information during the execution of the program, which eases the burden of static analysis.

Here we note that we assume that the statements of the same rank are examined according to their position in the program from beginning to end in both of the SNCM and the Tarantula methods. The percentage of the 130 versions in which the *T*-score are lower than or equal 1%, 10%, 20%, 30%,⋯, 100% are collected. The localization result of the two methods is shown in Figure 3.

From Figure 3, we can observe that SNCM performs better than Tarantula as a whole. However, Tarantula still performs better slightly than SNCM at the first 10% percentage. Particularly, SNCM can locate 13.54% faults when 10% of the codes are examined, while Tarantula can locate 14.35%. After that, SNCM outperforms Tarantula from 10% to 100%. We know that it hardly make sense for the examiner to examine too many codes in order to locate the faults, thus the first 50% stage is the emphasis to quantify the effectiveness of a localization method. It can be seen from Figure 3, SNCM can locate 35.96%, 47.25%, 59.46%, and 76.23% faults when 20%, 30%, 40%, 50% of the codes are examined, while that for Tarantula is 32.12%, 42.48% , 56.17 %, and 70.26%.

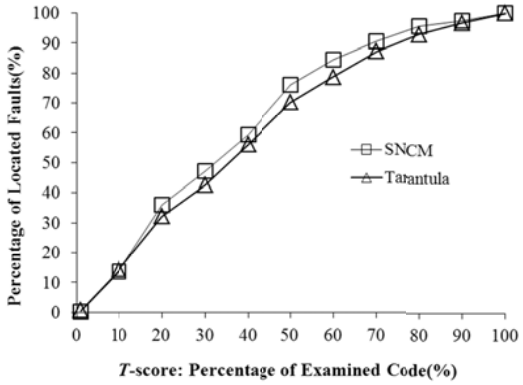


Figure 3. Cumulative Comparison between Tarantula and SNCM

We further discuss the influence of the experimentation paradigm with statistical repeatability adopted in this paper. Figure 4 gives the comparison of the results of Tarantula using the experimentation paradigm in the existing studies, which is called "non-repeatable" here, and the "repeatable" one in this paper.

Figure 4 shows that the performance of Tarantula is excellent using the non-repeatable paradigm, while that for the repeatable one is just average. As we discussed before, the non-repeatable paradigm runs the whole test suite to get the information for fault localization, which may not still be effective while test cases are input randomly according to certain testing profile. The comparison results in Figure 4 emphasis the importance of adopting the experimentation paradigm with statistical repeatability and the need to re-study the effectiveness of the existing methods.

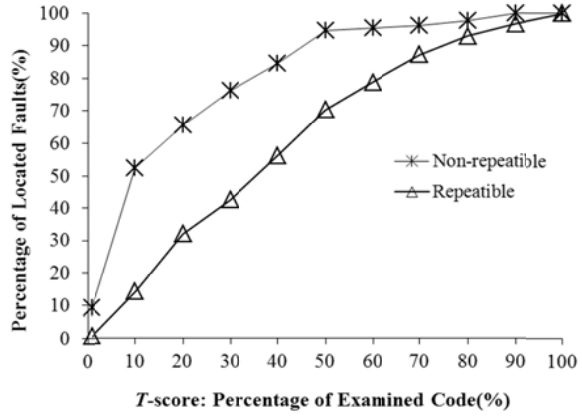


Figure 4. Comparison between the Performances of Tarantula using Different Experimentation paradigms

V. CONCLUSION

Many techniques have been proposed to help locate software faults. Although the existing methods have been reported to be effective by conducting some case studies, they have some obvious shortcomings. Inspired by the surprisingly discovery in the crossover studies of software and complex networks, a fault localization method based on software network centrality measures (SNCM) is proposed. The execution traces of the faulty trials and successful trials are modeled to be two networks, with the executed statements as nodes, and the execution from one statement to another as an edge. Two centrality measures (node degree and structural hole) are adopted to calculate the suspiciousness of each statement in both of the faulty trace network and the successful one. The statement which tends to have a higher centrality in the faulty network is more suspicious to be faulty than the other.

Modeling software from a network view can help handle the increasing complexity involved in fault localization which is a bottleneck of the development of fault localization techniques. Moreover, complex network views the complex systems globally, and the measures in complex networks focus on the macro level. The universal of the proposed method based on software network measures is thus promising. Besides, the experimental paradigm adopted in the existing methods cannot guarantee the repeatability of the localization effectiveness under random input according to a certain testing profile. The Tarantula method performs poorly under random input. Statistically repeatable experiment is applied to the subject programs in this paper. The experiment result shows that Tarantula still performs slightly better than SNCM at the first 10%. However, SNCM outperforms Tarantula after that from 10% to 100%.

Although this technique enhances the efficiency and effect of fault localization, there is still some work for us to do. Other centrality measures such as betweenness centrality, closeness centrality, and eigenvector centrality may be adopted by the SNCM method. These measures are widely used in network analysis to measure the centrality of a node within a network, which are essential and may further improve the method. In future work, another important one is that the experiment should be conducted on more program subjects to further guarantee the repeatability of the results.

REFERENCES

[1] X. P. Wang, Q. Gu, X. Chen, X. Zhang, D.X. Chen, "Research on Fault Localization Based on Execution Trace", Computer Science, 36(10): pp.168-171, 2009.

[2] A. Zeller, "Isolating Cause-effect Chains from Computer Programs", Proceedings of ACM symposia on Foundations of Software Engineering, pp.1-10, 2002.

[3] A. Zeller, R. Hildebrandt, "Simplifying and Isolating Failure-inducing Input", IEEE Tansactions on Software Engineering, Vol.28, No.2, pp.183-200, 2002.

[4] H. Cleve, A. Zeller, "Locating Causes of Program Failures", Proceedings of the 27th International Conference on Software Engineering, pp.342-351, 2005.

[5] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, M. I. Jordan., "Scalable Statistical Bug Isolation", Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, pp.15-26, 2005.

[6] H. Agrawal, R.A. DeMillo, and E.H. Spafford, "Debugging with Dynamic Slicing and Backtracking", Software: Practice & Experience, Vol.23, No.6, pp.589-616, 1996.

[7] W. E. Wong, Y. Qi, "Effective Program Debugging Based on Execution Slices and Inter-block Data Dependency", Journal of Systems and Software, Vol.79, No.7, pp.891-903, 2006.

[8] J.A. Jones, M.J. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-localization Technique", Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, pp.273-282, 2005.

[9] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization", Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing, pp.39-46, 2006.

[10] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi, "An Empirical Investigation of the Relationship Between Spectra Differences and Regression Faults", Journel of Software Testing, Verification and Reliability, Vol.10, No.3, pp.171-194, 2000.

[11] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, L. Yi, "Sober the Relationship Between Spectra Differences and Regression Faults", Journel of Software Testing, Verification and Reliability, Vol.10, No.3, pp.171-194, 2000.

[12] W. E. Wong, T. Wei, Y. Qi, L. Zhao, "A Crosstab-based Statistical Method for Effective Fault Localization", Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, pp.42-51, 2008.

[13] Kai-Yuan Cai, "Software Reliability Experimentation and Control", Journal of Computer Science and Technology, pp.697~707, 2006.

[14] A.de. Moura, Y.-C. Lai, A. Motter, "Signatures of Small-world and Scale-free Properties in Large Computer Programs", Physical Review E 68 (2) , 2003.

[15] C. Myers, "Software Systems as Complex Networks: Structure, Function, and Evolvability of Software Collaboration Graphs", Physical Review E 68 ,046116.1-046116.15, 2003.

[16] S. Jenkins, S.R. Kirk, "Software Architecture Graphs as Complex Networks: A Novel Partitioning Scheme to Measure Stability and Evolution", Information Sciences 177 , pp.2587-2601, 2007.

[17] K. Y. Cai, B. B. Yin, "Software Execution Processes as an Evolving Complex Network", Information Sciences, 179(12), pp.1903-1928, 2009.

[18] W. E. Wong, Y. Qi, L. Zhao, "Effective Fault Localization Using Code Coverage", Proceedings of The 31st IEEE Computer, Software, and Applications Conference (COMPSAC), 2007.

[19] R. S. Burt, Structural Holes: *The Social Structure of Competition, Harvard University Press*, pp.35-38, 1995.

[20] M. Renieris, S. Reiss, "Fault Localization with Nearest Neighbor Queries". Proceedings of the 18th IEEE International Conference Automated Software Engineering, pp.30-39, 2003.