World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:13, No:2, 2019

Bug Localization on Single-Line Bugs of Apache Commons Math Library

Cherry Oo, Hnin Min Oo

Abstract**—**Software bug localization is one of the most costly tasks in program repair technique. Therefore, there is a high claim for automated bug localization techniques that can monitor programmers to the locations of bugs, with slight human arbitration. Spectrum-based bug localization aims to help software developers to discover bugs rapidly by investigating abstractions of the program traces to make a ranking list of most possible buggy modules. Using the Apache Commons Math library project, we study the diagnostic accuracy using our spectrum-based bug localization metric. Our outcomes show that the greater performance of a specific similarity coefficient, used to inspect the program spectra, is mostly effective on localizing of single line bugs.

Keywords**—**Software testing, fault localization, program spectra.

I. Introduction

IDENTIFYING, localizing and repairing bugs are the vital activities of software development. While software testing forms the main activity for identifying program bugs, software repairing is the process of finding and correcting the buggy program portions. The bug localization process mentions to the problem of detecting buggy program portions given the failures of test execution. It has been recognized as one of the expensive parts of the repairing process, which justify the vital research effort for automated bug localization action [1].

Buggy statements in software code may lead the program failures such as crack or incorrect results and outcomes in the software development lifecycle. The task to decide and discover the buggy statements is called bug localization. In a software system, it will be very time consuming for the software developer to locate the buggy statements because of containing thousands of lines of code. Researchers have designed effective ways to find the buggy statement through bug localization approaches [1].

One of the popular in software repairing approaches is Spectrum-based Bug Localization (SBBL). In SBBL, the statement execution record (program spectra) of passing and failing test cases are examined to support program developers to locate the buggy statements. SBBL metrics have been designated to rank the buggy statements in program code according to their suspicious scores. In SBBL, statements with the highest score calculated by the SBBL metric will be ranked first as it is the most suspiciousness that might be the buggy statement. On the other hand, the statement with the lowest score is the safest statement as it is most not likely to be the buggy statement. Through this ranking, software developer can examine the top ranking statement first to locate the buggy statement rather than checking statement by statement from the beginning until the end of the program code.

The performance of SBBL metric is determined by how high it ranks the buggy statement based on the suspicious score calculated from the SBBL metric. In this paper, we analyze on single line bugs in Apache Commons Math Library project using our spectrum-based metric. In particular, the paper makes the following contributions:

- The first study is the comparison the bug-localization ability of our approach with Ochiai, Tarantula, and Jaccard. For the subjects studied, our study shows that, our approach consistently outperforms these techniques, performing it the best techniques known for bug localization on these subjects.
- The second study is a description of our approach in terms of suspicious ranking for their suspiciousness that provides a way to compare it with the Ochiai, Tarantula, and Jaccard techniques, as well as other future techniques [11].

The remaining of this paper is organized as follow: Section II outlines the background of Spectrum-based Bug Localization (SBBL), followed by methodology in Section III. We discussed our experimental results in Section IV and some related works in Section V and we concluded the paper in Section VI.

II. Preliminaries

A. Background

As input, a bug localization technique takes a buggy program and its test suite that contains at least one failing test, and as output, it produces a ranked list of suspicious statement locations, such as blocks or statements. In this paper, we use program statements as the locations [13].

Given a bug localization technique and a buggy program with a single-line buggy statement, a numerical measure of the quality of the technique can be computed as follows: (1) run the bug localization technique to compute the sorted list of suspicious program statements; (2) use a metric proposed in the literature to evaluate the effectiveness of a technique [13].

B. Failures, Errors, and Bugs

A program bug is a failure, error, fault, or flaw in a software that produces an unexpected or incorrect outcome. The bug repairing process regularly uses proper tools or techniques to identify bugs, and some computer systems find or repair

Cherry Oo is with University of Computer Studies, Mandalay, Myanmar (corresponding author, e-mail: cherryoo@ucsm.edu.mm).
Hnin Min Oo is with University of Computer Studies, Mandalay, Myanmar (e-mail: hninminoo@ucsm.edu.mm).

various bugs during operations since the 1950s.

Most of the bugs arise from errors and mistakes made in program source code, or program components. A small number of bugs are caused by compilers because incorrect code are produced by compilers. A buggy program can contain a large number of bugs that seriously interfere with its functionality. Bugs can affect errors that may have ripple effects. Bugs may have subtle effects or cause the program to freeze or crash the computer system.

C. Program Spectra

At run-time, program spectra are collected as the records that provide an exact observation on the lively behavior of program for different parts of a program, it classically consists of a number of flags or counters. In this paper, we work with statement hit spectra [2].

A hit spectrum of the program statement consists of a counter for every single statement of the program source code that indicates in a particular run whether or not that statement was executed [2].

D. Spectrum-Based Bug Localization

Two types of information are employed by the SBBL technique and they are gathered during program testing, clearly outcomes of testing and program spectra. While a program spectrum is a data collection, the testing outcome related with records whether each test case is failed or passed [16].

Given a buggy program P = {S1, S2, . . ., Sj} with j statements and executed by i test cases T = {T1, T2, . . . , Ti}. The testing outcomes of all test cases are recorded as spectra information of the program in form of a matrix. The component in the i^{th} row and j^{th} column of the matrix denotes the spectra information of statement Sj , by test case Ti , with 1 indicating Sj is executed, and 0 otherwise [16].

```
1 public int abs(int a, int b){
2           if (a > b) {
3                       return b - a;
4           }
5           return a - b;
6 }
```

Fig. 1 Buggy program example

TABLE I
TEST-SUITE FOR EXAMPLE BUGGY PROGRAM

Test Case	Input		Expected Output	Actual Output
	a	b		
Test1	3	5	-2	-2
Test2	5	3	-2	-2
Test3	4	0	4	-4
Test4	0	4	-4	-4
Test5	1	1	0	0

An example is shown in above. Fig. 1 shows a buggy program that contains six statements *{S1, S2, S3, S4, S5, S6}, but* we do not consider some statements that contain opening and closing curly bracket ({}). Table I shows five test cases *{T_1, T_2, T_3, T_4, T_5} to test the buggy program.* Specifically, four test cases among them pass *and* T_3 gives rise to *failed* run. The coverage information for each statement is recorded as a matrix. Finally, a coverage information matrix is generated by mean of the gathered information. The matrix is listed as Table II.

TABLE II
Coverage Information

Test Case	Statement				Error Status
	S_1	S_2	S_3	S_5	
T_1	1	1	0	1	0
T_2	1	1	0	1	0
T_3	1	1	1	0	1
T_4	1	1	1	0	0
T5	1	1	0	1	0

In Table II, S_i represents a statement of a buggy program; T_i represents a test case. (S_i, T_i)=1 represents S_i is covered by test case T_i; on the contrary, (S_i, T_i) = 0 represents S_i is not covered by test case T_i [15]. *ErrorStatus* denotes the program execution result of a test case, (*Error Status,* T_i) = 1 means the execution effect of T_i is fail whereas (*Error Status,* T_i)= 0 means pass. In addition, a_{11}, a_{10}, a_{01} and a_{00} are coverage statistics result of statement in program execution. For an execution of a statement with a test case, only one of these four symbols can be assigned by value 1, e.g, a_{11}=1 means this statement is covered by this test case and the result is fail. In Table I, a_{11}, a_{10}, a_{01} and a_{00} are the sum of the result value of a program execution with each test case respectively [14]. For example, (S_1, a_{10}) = 4 means S_1 is covered 4 times in total by test case set T = (T_1, T_2, T_3, T_4, T_5). For gathering aforementioned information accurately and rapidly, our study utilizes program instrumentation technique to obtain execution. Furthermore, one of the most popular unit testing tools- JUnit [20] is used to input test case.

Previous studies have identified some coefficients, such as Ochiai, and Tarantula, as the best metric to be used for SBBL. For example, a popular bug localization technique, Ochiai is defined as follows [2]

$$s_j = \frac{a_{11}(j)}{\sqrt{(a_{11}(j)+a_{01}(j))*(a_{11}(j)+a_{10}(j))}} \quad (1)$$

A program statement with higher suspicious value is a higher likelihood to be buggy. Therefore, the statements are sorted according to their suspiciousness in descending order after assigning the suspicious values to all program statements [10]. Repairing techniques begin from top to bottom of the ranking list. To identify the buggy statements, an effective method should be able as top in the ranking list as possible [16].

III. METHODOLOGY

SBBL techniques inform buggy statements only after examining a large number of lines or code elements. This section presents our approach to locate the buggy statements

from Apache Commons Math Project. There are two steps in our approach:

(1) Program spectrum information gathering. The statement coverage information and its execution result associated with certain test case set will be gathered.
(2) Calculate the suspicious value. For a program statement, the suspicious value is calculated by purposed bug localization method, respectively.

A. Apache Commons Math Library

The Apache Commons is a project of the Apache Software Foundation. The purpose is to provide reusable, open source Java software [8]. Commons Math is distributed under the terms of the Apache License, Version 2.0. Apache Commons Math consists of mathematical functions, structures representing mathematical concepts (like complex numbers, polynomials, vectors, etc.), and algorithms that we can apply to these structures (root finding, optimization, curve fitting, computation of intersections of geometrical figures, etc.). Apache Commons Math Library consists of 106 bugs in total. Among them, 26 bugs are single line bugs. We apply our spectrum-based ranking metric to localize the single line bugs in Apache Commons Math Library. In debugging process, we can repair single line bugs with some patterns such as, one line removal, one line addition, or one line replacement.

B. Program Spectrum Information Gathering

Program spectrum or coverage information reflects a certain face of a program execution. More specifically, coverage information shows whether a program unit is executed during execution with a certain test case. This information has been widely used in software testing, and it also can be used for fault localization. While program unit can be defined variously, such as statement, basic block, predicate, method and path, etc. In our study, statement coverage information is utilized since it is simple to calculate, and most important of all, the benefit of statement coverage is its ability to be used for statement-level bug localization. In addition, the corresponding execution result is also collected.

Our approach collected spectra information such as a11, a10, a01 and weight for each buggy statement while other SBBL techniques collected spectra information such as a11, a10, a01 and a00. We use weight value instead of a00 because the number of test cases is required and it is important whether each buggy statement is passed or failed by test cases [17]. So, we find that the weight values depend on

$$weighti = \{p \mid Sj = 1\} \tag{2}$$

In (2), p means the number of test cases which pass on each statement S_j. We calculate the suspiciousness of each buggy statement using the collected spectra information.

C. Calculating the Suspiciousness

Spectrum-based bug localization methods generally calculate the suspicious value by using collected information, such as a_{11}, a_{10}, a_{01} and a_{00} *(but we did not use this one)*. Researchers have proposed many formulas for calculating the suspicious value, and program units are ranked by the value to predict the probability of containing fault. Our SBBL metric is:

$$s_j = \frac{a_{11}(j)}{a_{11}(j) + a_{10}(j) + a_{01}(j) + weight_j} \tag{3}$$

In the above equation, a_{11} means the case which discovers bug when statement is passed. a_{10} means the case which does not discover bug when statement is passed. a_{01} means the case which discovers bug when statement is not passed and $weight_j$ means the number of test case which passes on each statement.

IV. Empirical Evaluation

Our experiments were performed on Intel(R) Core(TM) i3-6100U CPU @2.30GHz machine with 4.00 GB of RAM.

The effectiveness of SBBL technique is determined by the set of failed and passed test cases. Using two sets of test cases to locate, bugs may not be the most efficient approach [14]. We explore the following research questions:

RQ1: How the effectiveness of existing bug localization methods in the same program associated with test case set?

RQ2: How the effectiveness of our proposed method compared with some existing automatic bug localization methods.

We apply our SBBL technique to Apache Commons Math Library project, and check the output ranked list of single-line bugs identified as likely bug locations.

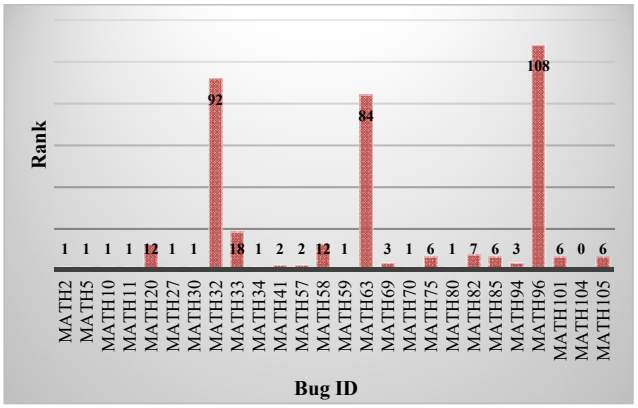

Fig. 2 Ranked list of single-line bugs using our metric

Fig. 2 shows the results of our method on single-line bugs from Apache Commons Math Library project. We made the localization of 26 single-line bugs by our method. We localized 10 out of 26 single-line bugs in rank one.

A. Evaluation Metric

For evaluating a bug localization technique, one important principle is to measure its effectiveness, such as statements that are inspected by programmers to locate the bugs. Also, a test set, when executed against the same program but in two altered environments, may result in two different sets of test cases [14]. To evaluate the effectiveness of our approach, we considered the following two metrics: mean reciprocal rank (MRR), and top N rank. MRR and top N rank are widely used

to evaluate bug localization techniques [7], [18].

Top-N: This metric counts the number of successfully localized within Top-N (N=1, 3, 5, 10) ranked results. If the bug localization techniques share the same score, we use the average position to present bug location. Higher Top-N denotes more effective bug localization [19].

Mean Reciprocal Rank (MRR): The reciprocal rank of a query is the reciprocal of the position for the first buggy statement in the results that is ranked as suspicious. MRR is the mean of the reciprocal ranks of the results of a set of statements, Q, and it can be calculated as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \quad (4)$$

MRR covers the overall quality of ranked suspicious statements. Larger values of all metrics indicate better accuracy [18].

B. Experimental Results

For RQ1, Fig. 2 shows the results of our proposed method. Both our metric and Ochiai metric got the top one position for 10 bugs, but Ochiai localized 12 bugs within top three and 17 bugs within top ten ranking list while our metric localized 14 bugs within top three and 19 bugs within top ten ranking list.

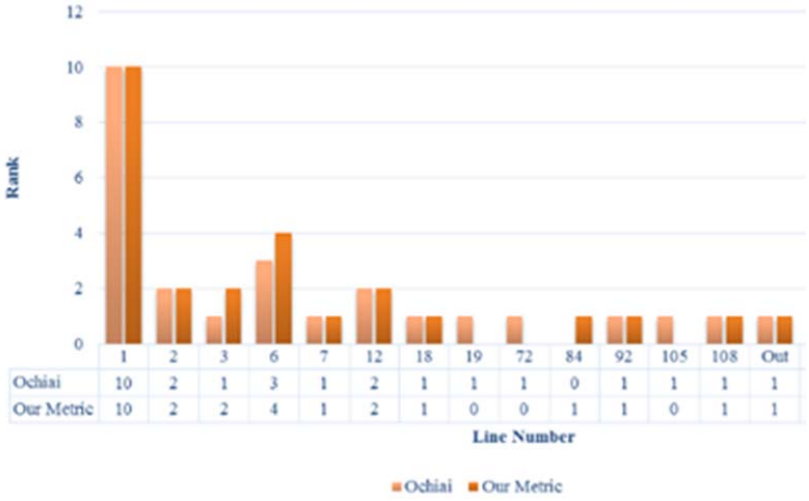

Fig. 3 Rank Level Comparison of Ochiai and Our Metric

TABLE III
Top-N and MRR Comparison with other approaches

Approach	Top-1 (%)	Top-3 (%)	Top-5 (%)	Top-10 (%)	MRR
Ochiai	38.5	46.2	46.2	65.4	0.47
Tarantula	0.00	3.85	3.85	7.69	0.04
Jaccard	0.00	0.00	0.00	11.54	0.03
Our Metric	**38.5**	**53.9**	**53.9**	**73.1**	**0.49**

For **RQ2**, Table III shows the results of our proposed method and other methods, i.e., Ochiai, Tarantula, and Jaccard [2], [3], [5], [9]. Both our metric and Ochiai metric produced for 10 bugs in Top-1, but Ochiai produced only **46.2%** average rate in Top-3 and **65.4%** in Top-10 level and got **0.47** in MRR while our metric produced **53.9%** in Top-3 and **73.1%** in Top-10 and got **0.49** in MRR. According to the results, our metric outperforms other metrics such as Ochiai, Tarantula and Jaccard. So, our approach is more effective than others in localizing for single-line bugs.

V. Related Work

Spectrum-based bug localization is the representative among the bug localization approaches [12]. Spectrum-based bug localization is the approach which estimates the relationship between the information about passed test cases failed test cases and the hit spectra information of statements [6]. If failed, test case occurs in the runtime, this case means that the statement contains a bug.

Spectrum-based bug localization means how to discover the bug location by using coverage information of a11, a10, a01, and a00. For example, it assumes that five test cases hit 3rd statements 3 times. If one test case among them is failed, this statement is likely to contain a bug relatively. However, if all test cases are passed, suppose that this statement is not likely to contain a bug. There are some representative algorithms such as Ochiai and Tarantula. Each algorithm calculates suspicious ratio respectively [15].

Abreu et al. [3] proposed a metric, called Ochiai, to get better effectiveness for bug localization techniques and then to enhance its diagnostic quality, they proposed a combination framework that is SBBL with a model-based debugging. The model-based approach is used for refining the ranking obtained from the spectrum-based method. Furthermore, Abreu et al. [4] also proposed a fault localization method to solve the multiple faults problem. For root cause analysis on the J2EE platform, Chen et al. [5] proposed a framework and it is targeted at large, dynamic Internet services, such as search engines and web-mail services. Jones et al. developed the Tarantula tool for the C language and works with spectra information [9].

In terms of early spectrum-based methods, only failed

information is utilized for locating bugs. Based on these methods, the later studies obtain the better results by means of using both the passing and failing test cases. SBBL method uses different metric to evaluate the probability of containing a bug of a unit of the program, and a ranking list is produced to highlight program units which strongly correlate with failures [16]. At present, many formulas of SBBL have already been proposed, typical SBBL methods include Ochiai, Jarccard, Tarantula, and so on.

VI. Conclusion

We conclude that the superior performance of our metric is in localizing single bugs in Apache Commons Math Library project. In this paper, we propose an effective spectrum-based bug localization for single-line Java bugs. We evaluated our approach on 26 real bugs. In our approach, we only need to study SBBL metrics from two collections (i.e., a11, a10). A statement executed by more failed test cases has higher possibility to be buggy so that it is observed to have the most significant outcome on the effectiveness of a metric. The experimental results showed that our approach outperforms existing three SBFL techniques significantly with low overhead.

In future work, we plan to study the localization of other single-line bugs, from large scale real-world Java programs.

References

[1] Abreu, R., Zoeteweij, P., Golsteijn, R. and Van Gemund, A.J., 2009. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82(11), pp.1780-1792.

[2] Abreu, R., Zoeteweij, P. and Van Gemund, A.J., 2006, December. An evaluation of similarity coefficients for software fault localization. In Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on (pp. 39-46). IEEE.

[3] Abreu, R., Zoeteweij, P. and Van Gemund, A.J., 2007, September. On the accuracy of spectrum-based fault localization. In Testing: Academic and Industrial Conference Practice and Research Techniques-Mutation (Taicpart-Mutation 2007) (pp. 89-98). IEEE.

[4] Abreu, R., Zoeteweij, P. and Van Gemund, A.J., 2009, November. Spectrum-based multiple fault localization. In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (pp. 88-99). IEEE Computer Society.

[5] Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A. and Brewer, E., 2002, June. Pinpoint: Problem determination in large, dynamic internet services. In null (p. 595). IEEE.

[6] Fu, W., Yu, H., Fan, G., Ji, X. and Pei, X., 2017, November. A Test Suite Reduction Approach to Improving the Effectiveness of Fault Localization. In Software Analysis, Testing and Evolution (SATE), 2017 International Conference on (pp. 10-19). IEEE.

[7] Gharibi, R., Rasekh, A.H. and Sadreddini, M.H., 2017, October. Locating relevant source files for bug reports using textual analysis. In Computer Science and Software Engineering Conference (CSSE), 2017 International Symposium on (pp. 67-72). IEEE.

[8] Hall, T., Zhang, M., Bowes, D. and Sun, Y., 2014. Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology (TOSEM), 23(4), p.33.

[9] Jones, J.A. and Harrold, M.J., 2005, November. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering* (pp. 273-282). ACM.

[10] Laghari, G., Murgia, A. and Demeyer, S., 2016, August. Fine-tuning spectrum based fault localisation with frequent method item sets. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 274-285). ACM.

[11] Le, T. D. B., Lo, D. and Li, M., 2015, September. Constrained feature selection for localizing faults. In 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp. 501-505). IEEE.

[12] Le, T. D. B., Oentaryo, R. J. and Lo, D., 2015, August. Information retrieval and spectrum based bug localization: better together. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp. 579-590). ACM.

[13] Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D. and Keller, B., 2017, May. Evaluating and improving fault localization. In Proceedings of the 39th International Conference on Software Engineering (pp. 609-620). IEEE Press.

[14] Schneidewind, N., Montrose, M., Feinberg, A., Ghazarian, A., McLinn, J., Hansen, C., Laplante, P., Sinnadurai, N., Zio, E., Linger, R. and Wong, E., 2010. IEEE Reliability Society Technical Operations Annual Technical Report for 2010. IEEE Transactions on Reliability, 59(3), pp.449-482.

[15] Wong, W.E., Qi, Y., Zhao, L. and Cai, K.Y., 2007, July. Effective fault localization using code coverage. In Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International (Vol. 1, pp. 449-456). IEEE.

[16] Xie, X., Chen, T.Y., Kuo, F.C. and Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering and Methodology (TOSEM), 22(4), p.31.

[17] Xu, Y., Yin, B., Zheng, Z., Zhang, X., Li, C. and Yang, S., 2019. Robustness of spectrum-based fault localisation in environments with labelling perturbations. Journal of Systems and Software, 147, pp.172-214.

[18] Youm, K.C., Ahn, J. and Lee, E., 2017. Improved bug localization based on code change histories and bug reports. Information and Software Technology, 82, pp.177-192.

[19] Zhang, M., Li, X., Zhang, L. and Khurshid, S., 2017, July. Boosting spectrum-based fault localization using PageRank. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 261-272). ACM.

[20] JUnit, http://www.junit.org.