Fault Localization Evaluation Framework

Yi Zhou
Donald bren school of information and computer sciences
Irvine, USA
zhouy46@uci.edu

Chiyu Cheng
Donald bren school of information and computer sciences
Irvine, USA
chiyuc2@uci.edu

***Abstract*—Testing is always a major step through the whole process of software engineering. In order to fix a bug, we must locate where the bug is. Traditional way to achieve such a goal is setting breaking point and find bug manually. It's a quite time-consuming method which is unrealistic to be applied in large-scale software system. In order to solve such problem, many automated fault-localization tools have been invented which could save developers lots of time for doing such tedious job. In this paper, we develop a framework to compare the efficiency of different fault-localization tools from several various dimensions.**

***Index Terms*—fault localization, software engineering, testing**

I. Introduction

Testing is one of the most critical and expensive tasks in the development of software. The process of detecting and locating bugs in a software system is called fault localization. Manually fault localization is acceptable for the small software system but unbearable for the large-scale system. Therefore, an automated fault localization technique is necessary for the real-world complex software system. There are many fault localization techniques like Tarantula, Jugular, and etc. However, there is no a comprehensive evaluation framework to evaluate and compare those techniques. An evaluation framework will guide the researchers to choose the right direction and method in the fault localization field. In addition, the process of evaluating different fault localization techniques will help us to understand this field better, and also help us to answer our research question - what is the most efficient, feasible, and accurate fault localization technique to locate the faults in the software system?

There are many challenges to implement this evaluation framework. The biggest challenge is how to build a runnable program of each fault localization technique we evaluate. It is impossible for us to re- implement the techniques by ourselves within only one month. Therefore, we are planning to use the existed code provided by the open source projects to test it. Another big headache is how to evaluate those techniques under the same standing since each technique has different context or definition. To solve this problem, we run those fault localization techniques on the same open source projects, and evaluate the results under our evaluation metric. To answer our research question, we evaluate several fault localization techniques including Tarantula, Ochiai, Barinel, Opt2, Jaccard, and Dstar2 on the defect4j faulty java projects, and compare their performance on our evaluation criteria including efficiency and accuracy. There several contributions of this paper: first, it provides a detailed evaluation metric to measure the effectiveness of fault localization techniques; second, our evaluation is scalable for the large dataset. Third, our testing dataset includes the multiple buggy position, which means our evaluation framework works for the real world software testing environment. In this paper, we first provide the background information in the field of fault localization and related previous work. Then we show our solutions to those challenges, and introduce the detailed methodology of our evaluation framework. In the end, we analyze the result of our evaluation framework to come up an conclusion to answer our research question, and discuss the threats to the validity of our results and the future direction of our evaluation framework.

II. Background Information

Most fault localization techniques take a faulty program as input and produce a ranked list of suspicious code locations at which program might be defective as output. Given one or more failing test cases and zero or more passing test cases, a FL technique outputs a list of suspicious program locations, such as lines, statements, or declarations. The FL technique uses heuristics to determine which program locations are most suspicious based on the score it generate.

There are two major forms of fault localization technique, one is Spectrum-based fault localization, which depends on statement execution frequencies. The more often a statement is executed by failing tests and the less often it is executed by passing tests the more suspicious the statement is considered. Another one is Mutation-based fault localization, which extends SBFL techniques by considering not just whether a statement is executed, but whether that statements execution is important to the tests success or failure, which means whether a change to that statement change the test outcome.

This paper considers 5 of the best-studied SBFL techniques. In the following, let *totalpassed* be the number of passed test cases and *totalpassed* be the number of those that executed statement s (similarly for *totalpassed* and *totalpassed*).

Method Name	Suspicious Score
Tarantula [6]	$S(s) = \frac{failed(s)/totalfailed}{failed(s)/totalfailed+passed(s)/totalpassed}$
Ochiai [7]	$S(s) = \frac{failed(s)}{sqrttotalfailed\cdot(failed(s)+passed(s))}$
Op2 [8]	$S(s) = failed(s) - \frac{passed(s)}{totalpassed+1}$
Barinel [9]	$S(s) = 1 - \frac{passed(s)}{passed(s)+failed(s)}$
Dstar [10]	$S(s) = \frac{failed(s)}{passed(s)+totalfailed-failed(s)}$

III. Related work

There are many metric to evaluate the effectiveness of a fault localization technique, such as LIL [2], T-score [3], Expense [4]. T-score uses the percentage of components that a software engineer would not have to examine before finding the first faulty position [3]. LIL is the technique that measures the performance of fault localization technique for the automated repair of software systems [2]. For the expense metric, it computes the percentage of executed statements that a software engineer would have to examine until find the first buggy position [4]. In this paper, we use different metric, which is similar with expense but with some changes. We focused on the efficiency, and accuracy. For the accuracy, we measure the percentage of all statements that a software engineer would have to examine until find the first and last buggy position.

IV. Empirical Methodology

In this section, we first introduce the evaluation criteria of our evaluation framework, and provide one example to show the process of evaluation for a fault localization technique on a small program. Then we show the detailed setup process to integrate our evaluation on the dataset defect4j.

A. *Evaluation Criteria*

In this paper, we use two main criteria, which includes efficiency and accuracy, to measure the effectiveness of a fault localization techniques. Efficiency means the amount of time that one technique spends to get the results. For the accuracy, there are two metrics, One is First, which measures the percentage of statements that have to be examined until the first buggy statement is reached, another one is Last, which measures the percentage of statements that have to be examined until the last buggy statement is reached. A lower score of a technique indicates the better performance. In the next part of this section, we provide an example to show how to compute the accuracy score.

B. *Evaluation Example*

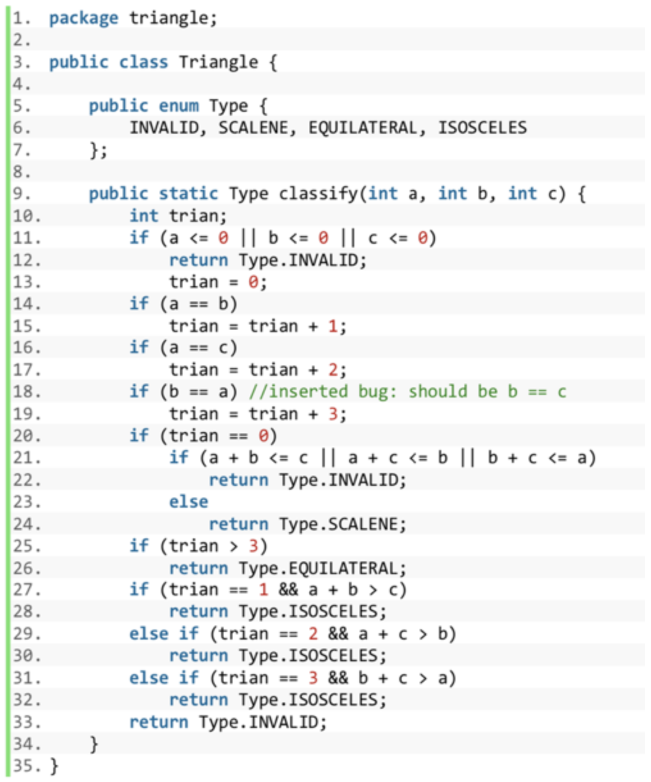
In this part, we evaluate the Ochiai technique on a small program named Triangle [5] to provides an example of the usage of our evaluation framework. Figure1 is the Triangle program, which is original bug-free, and we modify the line 18 to make it buggy. Figure 2 is the output of the Ochiai techniques, which is provided by the Gzolter toolset. The output contains a list of 25 potential buggy positions, and the real buggy line is 18, which is placed in the 10th of the output list. A developer has to inspect 10 statements to find the real buggy line if he/she uses the Ochiai techniques. Therefore, the accuracy score of Ochiai for the Triangle program is 10/25 = 0.4, and the score of First and Last is same in this example since there is only one buggy line exists in the Triangle program.

C. *Defect4j integration*

Due to the limited of computing resource, we are only be able to test the first 10 bugs of JFreeChart, Apache commons-lang, and Apache commons-math projects in the Defect4j open source dataset. The basic idea is similar with the previous triangle example. We follow the following steps to perform our experiments:

- Use the GZoltar toolset to collect the coverage information for one bug of the defec4j project.
- Collect the buggy line number of the defect4j project buggy line data corpus (online resource)
- Use the script based on the open source project[6] to compute the accuracy score of each technique, and save it in the CSV file.
- Use the python program parse_csvṗy to parse data and generate the plot for the score of each program.

For the detail of each step and testing environment, please check the readme.md file for more information.

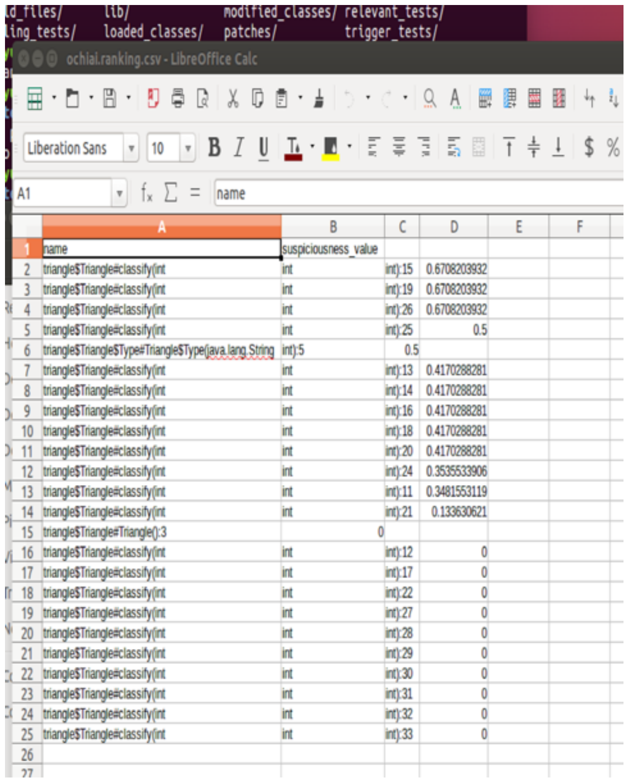


```
1.  package triangle;
2.
3.  public class Triangle {
4.
5.      public enum Type {
6.          INVALID, SCALENE, EQUILATERAL, ISOSCELES
7.      };
8.
9.      public static Type classify(int a, int b, int c) {
10.         int trian;
11.         if (a <= 0 || b <= 0 || c <= 0)
12.             return Type.INVALID;
13.             trian = 0;
14.         if (a == b)
15.             trian = trian + 1;
16.         if (a == c)
17.             trian = trian + 2;
18.         if (b == a) //inserted bug: should be b == c
19.             trian = trian + 3;
20.         if (trian == 0)
21.             if (a + b <= c || a + c <= b || b + c <= a)
22.                 return Type.INVALID;
23.             else
24.                 return Type.SCALENE;
25.         if (trian > 3)
26.             return Type.EQUILATERAL;
27.         if (trian == 1 && a + b > c)
28.             return Type.ISOSCELES;
29.         else if (trian == 2 && a + c > b)
30.             return Type.ISOSCELES;
31.         else if (trian == 3 && b + c > a)
32.             return Type.ISOSCELES;
33.         return Type.INVALID;
34.     }
35. }
```

Fig. 1: Sample Program

V. result

We evaluate six techniques, Tarantula, Ochiai, Barinel, Opt2, Jaccard, and Dstar2 on the three faulty program, JFreeChart, Apache commons-lang, and Apache commons-math. In this section, we show the performance of each techniques on each faulty project, and provide the overall performance of each techniques on all faulty projects in the end.



```
d_files/      lib/            modified_classes/ relevant_tests/
ing_tests/    loaded_classes/ patches/          trigger_tests/
```

ochiai.ranking.csv - LibreOffice Calc

A1 = name

	A	B	C	D	E	F
1	name	suspiciousness_value				
2	triangle$Triangle#classify(int	int	int):15	0.6708203932		
3	triangle$Triangle#classify(int	int	int):19	0.6708203932		
4	triangle$Triangle#classify(int	int	int):26	0.6708203932		
5	triangle$Triangle#classify(int	int	int):25	0.5		
6	triangle$Triangle$Type#Triangle$Type(java.lang.String	int):5	0.5			
7	triangle$Triangle#classify(int	int	int):13	0.4170288281		
8	triangle$Triangle#classify(int	int	int):14	0.4170288281		
9	triangle$Triangle#classify(int	int	int):16	0.4170288281		
10	triangle$Triangle#classify(int	int	int):18	0.4170288281		
11	triangle$Triangle#classify(int	int	int):20	0.4170288281		
12	triangle$Triangle#classify(int	int	int):24	0.3535533906		
13	triangle$Triangle#classify(int	int	int):11	0.3481553119		
14	triangle$Triangle#classify(int	int	int):21	0.133630621		
15	triangle$Triangle#Triangle():3	0				
16	triangle$Triangle#classify(int	int	int):12	0		
17	triangle$Triangle#classify(int	int	int):17	0		
18	triangle$Triangle#classify(int	int	int):22	0		
19	triangle$Triangle#classify(int	int	int):27	0		
20	triangle$Triangle#classify(int	int	int):28	0		
21	triangle$Triangle#classify(int	int	int):29	0		
22	triangle$Triangle#classify(int	int	int):30	0		
23	triangle$Triangle#classify(int	int	int):31	0		
24	triangle$Triangle#classify(int	int	int):32	0		
25	triangle$Triangle#classify(int	int	int):33	0		
26						
27						

Fig. 2: Sample Result

Based on our experiment, the result plot of JFreeChart shows the Opt2 is the most effective technique for the First score, and Dstart2 is the best for the Last score. The result plot of Math shows the Tarantula is the most effective technique for the First score, and Barinel is the best for the Last score. The result plot of Math shows the Tarantula is the most effective technique for the First score, and Barinel is the best for the Last score.

For the overall performance of each technique, the result plot indicates the Ochiai performs best for the First Score, and Ochiai/Jaccard performs equally good for the Last Score. And the average time for each program is around 15 minutes. Based on the score we collected from our experiments, there is no big difference between different techniques. In the overall slot, the difference of most techniques is less than 0.0001. The reason could be the insufficient amount of tests since we only evaluates the FL techniques on around 30 bugs. To conclude, the results of our experiments indicate that Ochiai is the most accuracy technique among the six evaluated fault localization techniques.

VI. Threats to Validity

There are two major threats that could affect the validity of our results from the previous section. One is the Efficiency evaluation metric; another one is insufficient experiments.

Based on our experiments, we believe that Efficiency is not an reasonable metric to measure the performance of a fault localization technique. We did all of our experiments on a MacBook with limited computing resource and overheated issue, which leads to the unstable CPU frequency when performing a long-term experiment. In addition, all of the techniques spend most of the time to collect the coverage information, which leads to the insignificant difference between the efficiency of each technique. In addition, we only evaluate the fault localization techniques on the first 10 bugs of each faulty programs. Our result might be biased because of that.

VII. Conclusion

Testing is one of the most critical and expensive tasks in the development of software. The process of detecting and locating bugs in a software system is called fault localization. Manually fault localization is acceptable for the small software system but unbearable for the large-scale system. Therefore, an automated fault localization technique is necessary for the real-world complex software system. There are many fault localization techniques like Tarantula, Jugular, and etc. However, there is no a comprehensive evaluation framework to evaluate and compare those techniques. An evaluation framework will guide the researchers to choose the right direction and method in the fault localization field. In this research, we propose a useful framework to evaluate several different fault-localization tools. We evaluate from the efficiency and the time each tool used to find faults. Due to the limit of computation source, this experiment is not perfect. However, we hope our evaluation framework could provide a useful thought for researchers and a reference for practitioners.

In the future, we will perform the fault localization techniques on all the bugs of the defect4j project, which will make our result more trust-worthy and comprehensive. In addition, there are different forms of fault localization techniques such as the mutation-based fault localization techniques. We will test more forms of fault localization to add more variance of our results.

References

[1] E. Wong, T. Wei, Y. Qi, and L. Zhao. A Crosstab-based Statistical Method for Effective Fault Localization. In Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, ICST 08, pages 4251, Washington, DC, USA, 2008. IEEE Computer Society.

[2] S. Moon, Y. Kim, M. Kim, and S. Yoo. Ask the mutants: Mutating faulty programs for fault localization. In ICST, pages 153162, Apr. 2014.

[3] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. Software Engineering, IEEE Transactions on, 32(10):831848, 2006

[4] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In ASE, pages 273282, Nov. 2005.

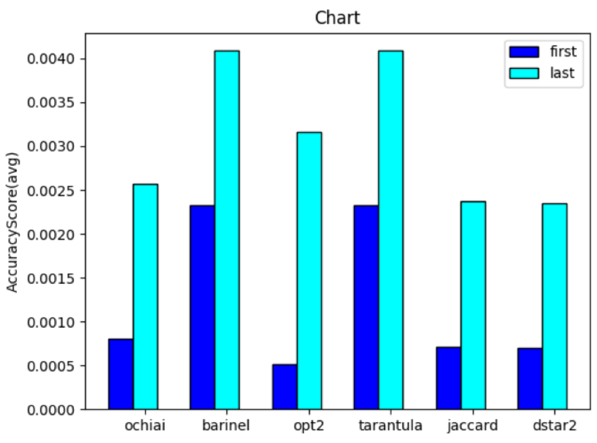
[5] Triangle https://gitlab.cs.washington.edu/kevinb22/fault-localization-research/ tree/master/src/triangle

[6] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In ASE, pages 273282, Nov. 2005

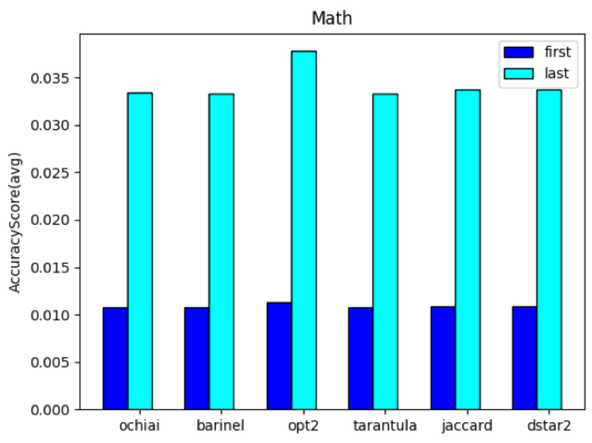
[7] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software, 82(11):17801792, 2009.

[8] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. ACM Transactions on software engineering and methodology (TOSEM), 20(3):11, 2011.

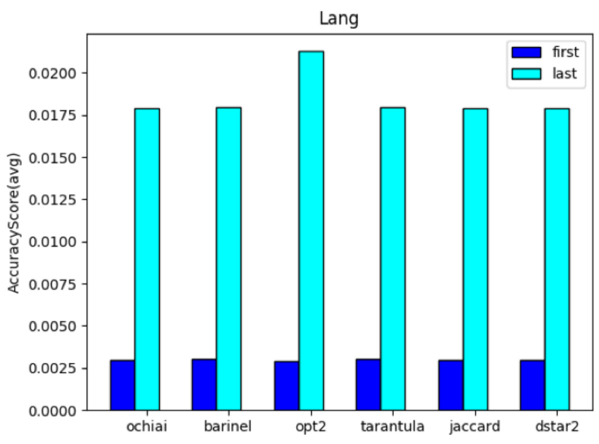
[9] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. Spectrum-based multiple fault localization. In Automated Software Engineering, 2009. ASE09. 24th IEEE/ACM International Conference on, pages 8899. IEEE, 2009.



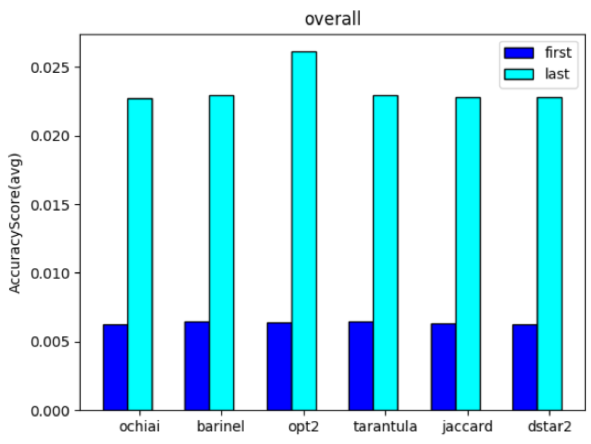
(a) Chart



(b) Math



(c) Lang



(d) Overall

Fig. 3: Final Results

[10] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. IEEE Trans. Reliab., 63(1):290308, Mar. 2014.