FSE:U:Estimating the Effectiveness of Spectrum-Based Fault Localization

Shuo Song
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
ss11@software.nju.edu.cn

ABSTRACT

Debugging is an important yet time-consuming activity to ensure software quality. To facilitate debugging, many automatic fault localization methods have been proposed. Spectrum-Based Fault Localization (SBFL) techniques make use of execution profile and testing results to calculate suspiciousness scores of program elements and output a list of program elements ordered accordingly. Developers can inspect program elements from the top of the list to identify faults. However, the effectiveness of SBFL depends on many factors, and the faulty program element may sometimes rank low in the ordered list. This may mislead developers and result in unnecessary manual work. In this paper, we present a preliminary study to estimate the effectiveness of SBFL before manual code walkthrough, so that we can decide whether to adopt SBFL for a given application. The method aims to avoid improper adoption of SBFL's results when they are actually misleading. In our method, we use two features, namely, Risk and Safety, to describe each program element. Our observation is that larger difference between the values of these two features indicates a more effective SBFL instance. We use the maximum difference of Risk and Safety among program elements as the estimator and find out a threshold value to indicate whether developers should adopt SBFL. We also propose a visualization method for our approach. Safety and Risk are mapped to color in RGB color model, so that developers can have a visual impression on the prediction of our method. We evaluate our approach via a preliminary empirical study using five real-world programs with a total of 90 faulty versions. The precision, recall and F-measure of our method are 80.5%, 95.1%, and 87.2% respectively, which show that our method can effectively predict the effectiveness of fault localization technique by identifying ineffective SBFL instances.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Fault Localization*

General Terms

Experimentation, Reliability

Keywords

Debugging, Fault localization, Risk formula, Estimator

1. PROBLEM AND MOTIVATION

Fault localization is one of the most time-consuming processes of debugging. Spectrum-Based Fault Localization (SBFL) uses various types of program spectrum and testing results to localize faults in a program [8]. SBFL techniques calculate suspiciousness scores of program elements and output a list of program elements ordered accordingly. They facilitate debugging by reduce the amount of code developers inspect to locate fault. Many empirical studies show the effectiveness of SBFL in debugging [7]. However, as a dynamic technique, the effectiveness of SBFL techniques depends on many factors, such as the pass-fail ratio of tests, the coverage of tests and the structure of the program under test [2][4][16]. In other words, the effectiveness of SBFL is not always guaranteed. Sometimes the faulty program element may rank very low in the list of program elements and thus mislead developers.[11][7] If the results of SBFL are misleading, it may lose the trust from developers as Parnin et al. suggested in their user study that developers did not find SBFL useful if the faulty element did not rank early in the list[12].

This is our motivation to estimate the effectiveness of SBFL prior to manual code walkthrough according to the localization result, such that we can decide whether to adopt SBFL and follow its result for the given scenario.

In this paper, we present a method to estimate the effectiveness of SBFL. In our method, we use statement as program element. For each statement, we define two features, namely Risk and Safety. Difference between the Risk and Safety values for each statement is calculated to estimate the effectiveness of SBFL. Interestingly, we have observed that when there exist cases where such difference is large, the SBFL technique tends to perform effectively. We also propose a visualization approach of our estimation, providing a more vivid presentation of the result of our method.

The effort of using our method to estimate the effectiveness of SBFL is trivial as compared to the effort of walking through source code with misleading SBFL results. We only use program spectrum and testing results collected during software testing and do not involve any complicated framework to do the prediction.

We have conducted empirical studies with five C programs to evaluate our method. Our experiment shows that we can predict the effectiveness of fault localization technique with a precision, recall and F-measure of 80.5%, 95.1%, and 87.2% respectively.

2. BACKGROUND AND RELATED WORK

Fault localization has been studied intensively and many methods have been proposed in the past decades [14]. Among all the methods, spectrum-based fault localization (SBFL) methods [8] have been widely studied. SBFL makes use of two kinds of information, namely testing results and program spectrum. Testing results are associated with each test case, recording testing results, e.g. passed or failed. Program spectrum is mainly the execution profile about program elements, such as statements, blocks and functions, regarding a specific test case. Execution profile records whether the test case executes a program element. For each program element s_i, the above information can be denoted as $< a^i_{ep}, a^i_{ef}, a^i_{np}, a^i_{nf} >$, where a^i_{ef} and a^i_{ep} denote the number of test cases executing s_i with the result of *fail* or *pass*, respectively and a^i_{nf} and a^i_{np} denote the number of test cases not executing s_i with the result of *fail* or *pass*, respectively. SBFL then applies risk formula on each program element. Many risk evaluation formulas are composed of a^i_{ep}, a^i_{ef} ,a^i_{np} and a^i_{nf}. For example, risk formula Tarantula[8] is defined as below.

$$\frac{\frac{a^i_{ef}}{a^i_{ef}+a^i_{nf}}}{\frac{a^i_{ef}}{a^i_{ef}+a^i_{nf}} + \frac{a^i_{ep}}{a^i_{ep}+a^i_{np}}} \tag{1}$$

Risk formulas assign each element with a suspiciousness score and a higher suspiciousness score indicates a higher possibility for the element to be faulty. SBFL then ranks program elements sorted by suspiciousness score in descending order. Developers can examine elements on top of the list to identify the faulty element and thus saves the effort of going through all source code.

However, the effectiveness of SBFL is affected by many factors and thus its result can sometimes be misleading. The empirical study by R. Abreu et al. indicated that the accuracy of SBFL would be affected by test cases used [2]. Their further study showed that a few (around 10) failed tests can be sufficient to reach near-optimal debugging effectiveness and the effect of passed tests is unpredictable [1]. N. DiGiuseppe et al. studied the effect of multiple faults on fault localization techniques [4]. Y. Yu et al.[16] studied the effects of test suite reduction on fault localization. Their results showed that the effectiveness of SBFL varies depending on the test suite reduction strategy used. B. Jiang et al. presented an empirical study to examine the impact of test case prioritization on the effectiveness of fault localization [6]. Y. Miao observed the effect of coincidental correctness on fault localization and proposed a clustering-based strategy to weaken the affects [11]. X. Xie provided a theoretical analysis of the suspicious formulas on SBFL [15].

All the above efforts study various factors to affect the effectiveness of SBFL. These studies show the effectiveness and the limitations of SBFL. We believe that SBFL can work well for some, but not all, application scenarios.

In [16], Y. Yu et al. assigned Tarantula suspiciousness and confidence metrics to each coverage entities, in which confidence value is to measure the degree of confidence in the given suspiciousness. The Tarantula confidence is greater if the entity is covered by more test cases. The Tarantula confidence is used in sorting coverage entities with same suspiciousness, instead of estimating SBFL effectiveness in the application as a whole.

Visualization was applied in the previous study of SBFL[8], but it was based on specific risk evaluation formula to highlight suspicious code, instead of predicting the effectiveness of SBFL.

3. APPROACH AND UNIQUENESS

As introduced in Section 2, risk evaluation formulas often combines a^i_{ep}, a^i_{ef}, a^i_{np} and a^i_{nf} to assign suspiciousness scores to program elements. Intuitively, if program element s_i is executed by most failed but few passed test cases, then s_i is very likely to be faulty.

In this paper, we propose an SBFL effectiveness estimator, which takes the program into consideration as a whole. The basic idea is to calculate the diversity of suspiciousness scores. Initially, we wanted to study the diversity of suspiciousness scores given by existed risk formulas by directly using statistical measures such as standard deviation and range. However, after empirical study and theoretical analysis, we found out that the standard deviation is strongly correlated with the adoption of risk formula. For example, the range of values calculated by *ER2* [15] is from 0 to 1 while the value of *ER1*[15] has no upper bound. Also, the number of failed and passed test cases exert an influence on standard deviation of suspiciousness scores, but studies [1] have found that a few failed tests can be sufficient to reach satisfying debugging effectiveness and the effect of the number of passed tests is unpredictable. Therefore, we cannot rely on statistical gauges of the output of risk evaluation formulas as a measurement. What we need is an estimator that is independent from specific risk evaluation formula and the absolute number of passed or failed test cases. Hence, we decide to trace back to the program itself and study its a^i_{ep}, a^i_{ef}, a^i_{np} and a^i_{nf}.

Therefore, for each program element s_i, we propose two features, namely *Risk* and *Safety*, defined as follows.

$$Risk_i = \frac{a^i_{ef} - min(a^{ef})}{max(a^{ef}) - min(a^{ef})} \tag{2}$$

$$Safety_i = \frac{a^i_{ep} - min(a^{ep})}{max(a^{ep}) - min(a^{ep})} \tag{3}$$

in which $max(a^{ef})$, $min(a^{ef})$, $max(a^{ep})$ and $min(a^{ep})$ are maximum a^{ef}_i, minimum a^{ef}_i, maximum a^{ep}_i and minimum a^{ep}_i for all program elements, respectively. $max(a^{ef}) - min(a^{ef})$ describes the range of a^i_{ef} for all program elements. Likewise, $max(a^{ep}) - min(a^{ep})$ describes the range of all a^i_{ep}. $Risk_i$ means the position of a^i_{ef} among all a_{ef} of program elements, regarding the $min(a^{ef})$. In the same way, $Safety_i$ means the position of a^i_{ep} among all a_{ep} of program elements, regarding the $min(a^{ep})$. By dividing the range, we normalize a^i_{ef} and a^i_{ep} to eliminate the influence of the absolute number of passed and failed test cases. The difference between $Risk_i$ and $Safety_i$, denoted as $Diff_i$ indicates the discrepancy between the normalized number of failed and passed test cases that executes element i. A higher $Diff_i$

	Test Suite 1										
	(1,3,1)	(2,3,1)	(1,3,2)	(1,1,1)	(1,1,2)	ef	ep	Risk	Safety	Diff	RGB
Pass/Fail	**Fail**	**Fail**	**Fail**	**Pass**	**Pass**						
1: if (x > y)	•	•	•	•	•	3	2	1	1	0	(255,255,0)
2: y = x;						0	0	0	0	0	(0,0,0)
3: if (y > z)	•	•	•	•	•	3	2	1	1	0	(255,255,0)
4: z = x;	•	•	•			3	0	1	0	1	(255,0,0)
5: return z;	•	•	•	•	•	3	2	1	1	0	(255,255,0)
max						**3**	**2**	1	1	**1**	
min						**0**	**0**	0	0	0	

Table 1: Example of Effective SBFL

	Test Suite 2										
	(1,3,1)	(2,3,1)	(1,3,2)	(3,3,2)	(3,2,1)	ef	ep	Risk	Safety	Diff	RGB
Pass/Fail	**Fail**	**Fail**	**Fail**	**Pass**	**Pass**						
1: if (x > y)	•	•	•	•	•	3	2	1	1	0	(255,255,0)
2: y = x;					•	0	1	0	0.5	-0.5	(0,127,0)
3: if (y > z)	•	•	•	•	•	3	2	1	1	0	(255,255,0)
4: z = x;	•	•	•	•	•	3	2	1	1	0	(255,255,0)
5: return z;	•	•	•	•	•	3	2	1	1	0	(255,255,0)
max						**3**	**2**	1	1	**0**	
min						**0**	**0**	0	0.5	-0.5	

Table 2: Example of Ineffective SBFL

indicates that more failed and less passed test cases execute element s_i.

We calculate the estimator of SBFL effectiveness by performing the following steps. First, for each statement e_i, we calculate $Risk_i$, $Safety_i$ and $Diff_i$. Next, we pick the maximum $Diff_i$, denoted as $Estimator = max(Diff_i)$, among all program elements as an estimator. The range of $Risk$ and $Safety$ are both 0 to 1, so it is easy to show that the range of $Estimator$ is from -1 to 1. Intuitively speaking, large $max(Diff_i)$ indicates the existence of at least one element that is intensively executed by large portion of the failed test case. On the other hand, small $max(Diff_i)$ means passed failed test cases execute all program elements quite evenly. Interestingly, we have found that scenarios with large $max(Diff_i)$ tends to provide an effective SBFL results; while small $max(Diff_i)$ is usually a sign of ineffectiveness. The $Estimator$ is comparable among different programs. Empirically, we set the threshold value of $max(Diff_i)$ indicating SBFL effectiveness to be 0.5. Values larger than 0.5 means developers can choose to use SBFL to facilitate debugging and otherwise not to choose it.

We also propose a visualization method to display our estimation result. We colorize statements in RGB model. Intuitively, we use Red to present Risk and use Green to express Safety, and Blue is left to 0. Based on the RGB model, the Risk and the Safety values are normalized to 0 to 255 as follows.

$$Red_i = 255 \times Risk_i \tag{4}$$

$$Green_i = 255 \times Safety_i \tag{5}$$

Benefits of such visualization are to provide vivid information and a straightforward estimation process. In other words, if the programmer can view some statement with vivid red color, he/she can go ahead with the code inspection by following the given SBFL localization results. Table 1 shows an example of $Estimator$ calculation. In table 1, statement line 2 is colored in vivid red, which indicates effective SBFL result. On the other hand, if the Diff is very small (i.e. all units have low R values and high G values), there is no vivid red statement. Based on our empirical studies, SBFL tends to give worse performance in such a case, and the programmer is suggested to consider other fault localization techniques. Table 2 shows an example of such case, where there is no statement in vivid red.

Only few previous studies focus on the prediction of SBFL effectiveness. Le et al.[9] built an oracle that leverages machine learning techniques to identify features related to the effectiveness of SBFL aiming to predict the effectiveness of SBFL. They presented a method to predict the effectiveness of a given SBFL scenario. Different from our proposal that explicitly indicates the signal of being effective, they used a supervised machine learning model to learn the prediction result from a training set. As compared to their method, our proposal is more lightweight. Moreover, with the assistance of visualization strategy, we can demonstrate the signal of being effective or ineffective vividly, and hence help users to make prediction more quickly and easily.

4. RESULTS AND CONTRIBUTION

4.1 Dataset

We use five C programs from Software-artifact Infrastructure Repository (SIR) [5] to test our method. These programs have been used in previous studies on fault localization[2] [7] [13] [10] and therefore can be used as experimental benchmark. Along with source code, SIR also provides

Program	LOC	#Faulty	#Tests	Description
schedule2	374	9	2710	priority scheduler
print_tokens	726	5	4130	lexical analyzer
print_tokens2	570	10	4115	lexical analyzer
replace	564	31	5542	pattern replacement
space	6199	35	13585	interpreter

Table 3: Datasets

seeded version with faults and test suites. We use *Gcov* [1], a code coverage testing tool conjuncted with GCC to make statement-level instrumentation and execute the program under all test cases to construct the program spectrum for fault localization. Table 3 presents the details of these programs.

Among the above five programs, four of them, namely *schedule2*, *print_tokens*, *print_tokens2* and *replace* are Siemens programs, which were created by Siemens Corporate Research for a study on test coverage criteria. Each of the programs has seeded faulty versions and each faulty version contains one fault that can span one or more statements. We exclude versions with variable declaration bugs because our instrumentation cannot reach declaration statements. *space* was created as an interpreter used by European Space Agency. We also exclude three versions that are semantically equivalent to the bug-free version and used 35 faulty versions. In total, we include 90 faulty versions from five programs.

4.2 Evaluation Metrics

Our method calculates Safety and Risk value of program elements and then presents a suggestion of whether SBFL should be adopted. We compare the output of our estimating method with the effectiveness of SBFL, which is measured by *Expense*, the percentage of the program that must be examined to identify the fault according to the ranked list of statement sorted by suspiciousness score provided by SBFL [16]. A higher *Expense* means more source code of the program should be examined before locating the fault and thus SBFL is less effective. *Expense* is computed by the following equation.

$$Expense = \frac{rank\ of\ faulty\ statement}{number\ of\ executable\ statements} \times 100\% \quad (6)$$

As for SBFL formula, we use a popular risk formula *Jaccard* [3] to locate the seeded faults. *Jaccard* is computed as $\frac{a^i_{ef}}{a^i_{ef}+a^i_{nf}+a^i_{ep}}$.

We then evaluate the estimating method in terms of precision, recall and F-measure[9]. The method estimates the effectiveness of SBFL by identifying ineffective fault localization. The concept of true positives, false positives, true negatives and false negatives are shown as follows:

True Positives(TP): Number of ineffective SBFL instances that are predicted correctly

False Positives(FP): Number of effective SBFL instances that are predicted wrongly

True Negatives(TN): Number of effective SBFL instances that are predicted correctly

False Negatives(FN): Number of ineffective SBFL instances that are predicted wrongly

[1]https://gcc.gnu.org/onlinedocs/gcc/Gcov.htm

Based on these concepts, the definition of precision, recall and F-measure are as follows:

$$Precision = \frac{TP}{TP + FP} \quad (7)$$

$$Recall = \frac{TP}{TP + FN} \quad (8)$$

$$F - measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (9)$$

Both precision and recall are important. A low precision means that many of SBFL instances that are estimated as ineffective are actually effective while a low recall means that many ineffective instances are wrongly estimated to be effective. However, there is often a trade-off between precision and recall. F-measure is often used to assess the trade-off between precision and recall. It is the harmonic mean of precision and recall that combines the two as a single summary measure.

4.3 Results

We conduct the experiment on 90 faulty versions from five programs written in C. Empirically, if *expense* is over 10%, meaning that developers have to read through more than 10% of the source code, then SBFL is defined as ineffective. We observe that 0.5 can be the threshold value of *Estimator* to estimate the effectiveness of SBFL .

Based on the above results and the definition of precision, recall and F-measure, we calculate the measures and present them in Table 4. The last row is the average precision, recall and F-measure values calculated based on all five programs.

Program	Precision	Recall	F-measure
schedule2	83.3%	100%	90.8%
print_tokens	80%	100%	88.9%
print_tokens2	80%	100%	88.9%
replace	75.5%	89.2%	81.9%
space	70%	100%	86.0%
Average	80.5%	95.1%	87.2%

Table 4: Recision, Recall and F-measure

The above result shows that we can achieve an average precision of 80.5%, recall of 95.1% and F-measure of 87.2%. The high recall value means that we can identify most of the ineffective SBFL instances using our estimating method, thus reducing unnecessary code walkthrough efforts by 95.1%. The precision value indicates that 19.5% ineffective faulty versions are estimated wrongly, meaning that they are in fact suitable for SBFL but our method suggests the opposite. However, developers can leverage many other approaches to improve program comprehension and make use of fault localization methods other than SBFL to facilitate debugging.

The F-measure is used to measure the trade-off of precision and recall. Our F-measure is 87.2%, and is quite comparable with other studies on prediction tasks in software engineering [9].

5. CONCLUSION AND DISCUSSION

In this paper, we propose a lightweight method to predict the effectiveness of SBFL, so as to reduce unnecessary code inspection if SBFL results are misleading. We find out two features, $Risk$ and $Safety$, that are related to the effectiveness of SBFL and use the difference of the two features to estimate SBFL effectiveness. The estimation of our method can also be represented in color to provide vivid information and a straightforward estimation process. We also find out a threshold value to indicate whether developers should adopt SBFL or not. We verify our method via an experiment using five real-world programs with a total of 90 faulty versions. The precision, recall and F-measure of our method are 80.5%, 95.1%, and 87.2% respectively, which shows that our method could effectively predict the effectiveness of fault localization technique by identifying ineffective SBFL instances.

As we discussed above, the effectiveness of SBFL may be affected by many factors. Hence, we should design more sufficient estimator and design more comprehensive empirical studies to verify our approach in the future. We should involve more programs with different sizes and different structures in different programming languages to verify our approach in the future. We will also use test suites, with different passed tests and failed tests for the same program to investigate the impact of test suites. We use *Jaccard* as the risk formula of SBFL in this paper, more experiments should be conducted to find out the correlation between our approach with other risk evaluation formulas. We also believe the fault types will affect the effectiveness of SBFL techniques. All the factors will be considered in our future work.

6. ACKNOWLEDGMENT

The author would like to thank Dr. Zhenyu Chen, Dr. Xiaoyuan Xie, and Zicong Liu for their valuable comments. This work is partially supported by the National Natural Science Foundation of China (61170067, 61373013).

7. REFERENCES

[1] 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):1780–1792, 2009.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[3] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 595–604. IEEE, 2002.

[4] N. DiGiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 210–220. ACM, 2011.

[5] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[6] B. Jiang, Z. Zhang, T. Tse, and T. Y. Chen. How well do test case prioritization techniques support statistical fault localization. In *Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International*, volume 1, pages 99–106. IEEE, 2009.

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

[8] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477. ACM, 2002.

[9] T.-D. B. Le, D. Lo, and F. Thung. Should i follow this fault localization tool's output? *Empirical Software Engineering*, pages 1–38, 2014.

[10] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[11] Y. Miao, Z. Chen, S. Li, Z. Zhao, and Y. Zhou. A clustering-based strategy to identify coincidental correctness in fault localization. *International Journal of Software Engineering and Knowledge Engineering*, 23(05):721–741, 2013.

[12] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[13] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 30–39. IEEE, 2003.

[14] W. E. Wong and V. Debroy. A survey of software fault localization. *Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45-09*, 2009.

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

[16] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proceedings of the 30th international conference on Software engineering*, pages 201–210. ACM, 2008.