Improving the Accuracy of Spectrum-based Fault Localization for Automated Program Repair

Tetsushi Kuma, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
{t-kuma,higo,shinsuke,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

The sufficiency of test cases is essential for spectrum-based fault localization (in short, SBFL). If a given set of test cases is not sufficient, SBFL does not work. In such a case, we can improve the reliability of SBFL by adding new test cases. However, adding many test cases without considering their properties is not appropriate in the context of automated program repair (in short, APR). For example, in the case of GenProg, which is the most famous APR tool, all the test cases related to the bug module are executed for each of the mutated programs. Execution results of test cases are used for checking whether they pass all the test cases and inferring faulty statements for a given bug. Thus, in the context of APR, it is important to add necessary minimum test cases to improve the accuracy of SBFL. In this paper, we propose three strategies for selecting some test cases from a large number of automatically-generated test cases. We conducted a small experiment on bug dataset Defect4J and confirmed that the accuracy of SBFL was improved for 56.3% of target bugs while the accuracy was decreased for 17.3% in the case of the best strategy. We also confirmed that the increase of the execution time was suppressed to 1.5 seconds at the median.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

KEYWORDS

Fault Localization, Debug, Test Generation

ACM Reference Format:
Tetsushi Kuma, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. 2020. Improving the Accuracy of Spectrum-based Fault Localization for Automated Program Repair. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3387904.3389290

1 INTRODUCTION

Fault Localization (in short, *FL*) is one of the well-known and well-researched techniques to support debugging. FL infers the locations of defects in a given buggy program. Various FL methods have been proposed so far [6, 8, 12]. *Spectrum-Based Fault Localization* (in short, *SBFL*), which is a kind of FL based on execution paths of test cases, is one of the most actively studied methods in recent years [14]. The basic idea of SBFL is that program statements executed in failed test cases are likely to be defective, and program statements executed in passed test cases are unlikely to be defective. SBFL receives a buggy program and its test cases as inputs, and then it infers faulty statements for the defects using the information on the pass/failure of each test case and their spectra. The accuracy of SBFL depends on the test cases given as inputs because SBFL performs localizing defects using only information obtained by executing test cases. Consequently, some studies focused on test cases to improve the accuracy of SBFL. Dandan's research [4] and Li's research [11] show selecting some test cases from original ones improves SBFL. However, their approach cannot be applied when the number of original test cases is small.

Subject program		Test cases			
		t1	a1	a2	a3
1	`public class IEEE754rUtils {`				
2	`  float max(float a, float b) {`				
3	`  if(Float.isNaN(a))`	●	●	●	●
4	`  return a; //return b;`	●	●		
5	`  else if(Float.isNaN(b))`	●	●	●	●
6	`  return a;`	●	●	●	
7	`  else`	●	●		●
8	`  return Math.max(a, b);`	●	●		●
9	`  }`				
10	`  float max(float[] array) {`				
11	`  ...`				
12	`  float max = array[0];`	●	●	●	●
13	`  for(int j = 1; j < array.length; j++)`	●	●	●	●
14	`  max = max(array[j], max);`	●	●	●	●
15	`  return max;`	●	●	●	●
16	`  }`				
17	`}`				
Pass/Fail status		F	F	P	P
Number of faulty candidates		10	10	3	1

Figure 1: An example of adding test cases that improve FL

Figure 1 is a part of the program included in Apache Commons Lang (in short, Lang)[1]. In the program, the fourth line includes a defect. The fourth line is '`return a;`', but it must be '`return b;`'. If test case `t1`, which is included in `Lang`, is executed for this program, it fails. When an SBFL method is applied to the program, the results indicate all statements executed by `t1` are suspicious. In this example, many lines of code are regarded as highly possible to be defective, which will not help developers to localize the defect. Thus, in this example, SBFL is not helpful if we use only test cases attached to the target project, and it is impossible to select test cases to improve SBFL because `t1` is the only one for this program.

The authors consider that the accuracy of SBFL can be improved if additional test cases are also used with the original test cases. In Figure 1, we use three new test cases `a1`, `a2`, and `a3`, in addition to

[1]Some methods and statements have been omitted to simplify the explanation.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389290

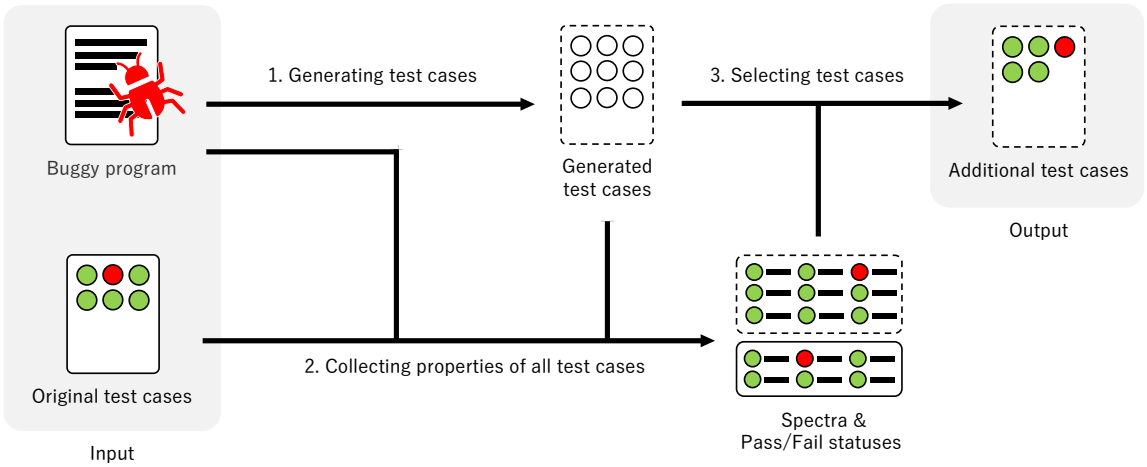

Figure 2: Overview of the proposed technique

t1. When the same SBFL method is applied by using those four test cases, the fourth line is regarded as the most suspicious statement. However, in this example, we have to see the following facts:

- adding a2 and a3 can improve localizing the defect, but
- adding a1 does not contribute to the improvement.

This is because a2 and a3 have different spectra from the others, whereas a1 has the same spectrum as t1, and it has only the same information as t1. Thus, even if a test case like a1 is added, it does not affect the accuracy, but only increases the execution time.

It is necessary to consider not only the accuracy but also the execution time of SBFL when SBFL is used in the context of *Automated Program Repair* (in short, *APR*). APR is a technique that automatically alters a given buggy program to remove defects. In APR, SBFL is used to determine where to alter the program. It is important to improve the accuracy of SBFL from the viewpoint of selecting the faulty statements for altering. In generation & validation techniques of APR such as GenProg [10], all test cases are executed for each of the mutated programs to validate them and to perform SBFL. Thus, if test cases not contributing to improving SBFL are used in APR, APR tools finish program repairs with unnecessarily-longer time or fail to repair programs due to time constraint given by users. Consequently, it is important to add only test cases contributing to improving SBFL in the context of APR.

In this research, we are trying to develop a technique to obtain test cases each of which contributes to improving SBFL. The proposed technique generates test cases for a buggy program and selects test cases that have different spectra from original test cases. Removing test cases not improving FL prevents unnecessary increase in the execution time. As an evaluation of the proposed technique, we compared the results of applying the SBFL with the proposed technique and the results of applying the SBFL only with the original test cases. Those results showed that the accuracy of SBFL was improved or the same for 82.7% of the faulty statements for the target defects. The increase in execution time with the proposed technique was suppressed to 1.5 seconds at the median.

2 SPECTRUM-BASED FAULT LOCALIZATION

Spectrum-based fault localization (in short, SBFL) is a kind of FL, which is the technique to infer the locations of defects in a buggy program. The input of SBFL is a buggy program and its test cases. SBFL executes all test cases and collects the spectrum and the pass/fail status of each test case. A spectrum of a test case is a set of program statements with execution flags. A flag indicates whether the test case covers the statement. Based on the spectra and the pass/fail statuses, SBFL calculates the suspiciousness for all statements and outputs it. Many methods to calculate suspiciousness have ever been proposed. According to Abreu, Ochiai [3] is the best method to calculate suspiciousness in the compared seven methods [1]. Ochiai calculates suspiciousness on the following formula (1).

$$susp(s) = \frac{fail(s)}{\sqrt{totalfail \times (fail(s) + pass(s))}} \quad (1)$$

Each variable in the formula shows the following element:

s: a statement,
$totalfail$: the number of all failed test cases,
$fail(s)$: the number of failed test cases which cover s, and
$pass(s)$: the number of passed test cases which cover s.

3 PROPOSED TECHNIQUE

To improve the accuracy of FL while suppressing the increase of execution time, we propose a technique to obtain test cases whose spectra are different from any of the original ones. Adding such test cases enables to calculate more accurate suspiciousness for each statement. Figure 2 shows an overview of the proposed technique. The input of the technique is a buggy program and its original test cases, and the output is test cases used for FL in addition to original ones. The proposed technique consists of the following three steps:

Step1: generating test cases,
Step2: collecting the properties of all test cases, and
Step3: selecting test cases for FL in addition to original ones.

3.1 Generating test cases

The candidate test cases that are added to the original ones are generated by an automated test generation tool. There are two ways to generate test cases by automated test generation tools:

- generating the input values by a tool, but specifying the expected values manually and

- generating both the input and expected values by a tool.

In the former, the input values are generated from a buggy program, but we need to understand the program behavior to specify the expected ones. The latter generates test cases without human effort, but it requires a non-buggy program to generate the expected ones. Automatically generating the expected values is challenging, known as the oracle problem [2]. Some automated test generation tools regard the outputs of the input program as the expected values. In this research, we generate both the input and expected values by a tool to reduce the cost of test generation. A program before defects occur is used as a non-buggy program.

3.2 Collecting properties of all test cases

All original test cases and all generated test cases are executed on the buggy program. We collect the spectrum and the pass/fail status as the properties of each test case.

3.3 Selecting test cases

We select test cases based on particular criteria from all test cases generated in Step1 with the properties collected in Step2. Removing test cases which do not improve FL prevents increasing the execution time unnecessarily.

Our basic approach to select test cases considers the spectra. Each of the generated test cases is checked whether its spectrum is different from any other test cases. If so, it is selected. It is important to get new information from additional test cases for improving FL. The new-spectrum test cases enable to make a difference between the statements with the same suspiciousness. In the motivating example, the same-spectrum test cases do not improve FL, but in some cases, they may contribute to imoroving FL. For example, the test case that has the same spectrum as any of the original ones, but its pass/fail status is different from them also has new information. Such test cases correct false negative test results, that is, correct suspiciousness for non-faulty statements with high suspiciousness and faulty ones with low suspiciousness. Failed test cases may also contribute because Kucuk's research [9] indicates that SBFL tends to perform poorly if the number of failed test cases is small. Thus, we try the following three strategies:

Strat1: selecting new-spectrum test cases,
Strat2: selecting new-spectrum test cases or the same-spectrum test cases with a different pass/fail status, and
Strat3: selecting new-spectrum test cases or failed test cases.

Figure 3 is an example of selecting test cases[2]. t1 is the original test case, and c1, c2, c3, and c4 are generated ones in Step1. Each of test cases c1, c2, c3, and c4 shows the following properties:

c1: having a new spectrum and its status is 'passed',
c2: having the same spectrum as t1 and its status is 'failed',
c3: having a new spectrum and its status is 'failed', and
c4: having the same spectrum as c3 and its status is 'passed'.

Consequently, c1 and c3 are selected with Strat1, c1, c3, and c4 are selected with Strat2, c1, c2, and c3 are selected with Strat3.

[2]the subject is the same as Figure 1, but the test cases are changed for explanation

	Subject program	Test cases				
		t1	c1	c2	c3	c4
1	public class IEEE754rUtils {					
2	float max(float a, float b) {					
3	if(Float.isNaN(a))	●	●	●	●	●
4	return a; //return b;	●		●	●	●
5	else if(Float.isNaN(b))	●	●	●	●	●
6	return a;	●		●	●	●
7	else	●	●	●		
8	return Math.max(a, b);	●	●	●		
9	}					
10	float max(float[] array) {					
11	...					
12	float max = array[0];	●	●	●	●	●
13	for(int j = 1; j < array.length; j++)	●	●	●	●	●
14	max = max(array[j], max);	●	●	●	●	●
15	return max;	●	●	●	●	●
16	}					
17	}					
	Pass/Fail Status	F	P	F	F	P
Add/Not (Add=1,Not=0)	Strat1	-	1	0	1	0
	Strat2	-	1	0	1	1
	Strat3	-	1	1	1	0

Figure 3: An example of selecting test cases

4 EXPERIMENT

We conducted a small experiment on an open source project with the proposed technique. The purposes of our experiment are answering the following two questions.

RQ1: How much can our technique improve the accuracy?
RQ2: How much can our technique suppress the execution time, compared with using all the generated test cases?

4.1 Subject program

The subject program of our experiment is Apache Commons Math (in short, Math) in Defects4J [7]. Defects4J is a dataset that provides real bugs that occurred in the development of several open source projects. It has the information which test cases are failed and where faulty statements are. Math contains 106 bugs, and all of them are used in our experiment. The reason why we use Math is that it is often used as a benchmark in papers on FL [13].

4.2 Experimental setup

We generated test cases by using EvoSuite [5]. EvoSuite is an automated test generation tool and generates test source code for given Java classes. We used default values for all the EvoSuite parameters. Thus, the time budget to generate test cases was 60s. By using those test cases, we made the following five sets of test cases:

S_1: original test cases + test cases selected with Strat1,
S_2: original test cases + test cases selected with Strat2,
S_3: original test cases + test cases selected with Strat3,
S_{all}: original test cases + all generated test cases, and
S_{ori}: only original test cases.

Those test cases are different depending on bugs in Math because they test buggy classes, which depend on bugs. The average numbers of test cases added to the original ones in S_1, S_2, S_3 and S_{all} were, 5.5, 6.0, 5.7, and 37.4 per class, respectively. We compared the accuracy and the execution time of FL by using those test cases.

In this experiment, the rank of the faulty statements in the ranking according to the suspiciousness was used as an evaluation metric for the accuracy of FL. If the suspiciousness of some statements were the same, we considered the rank of the faulty statement was the worst rank. For example, if the suspiciousness of the faulty statement is the third largest, but other two statements also have

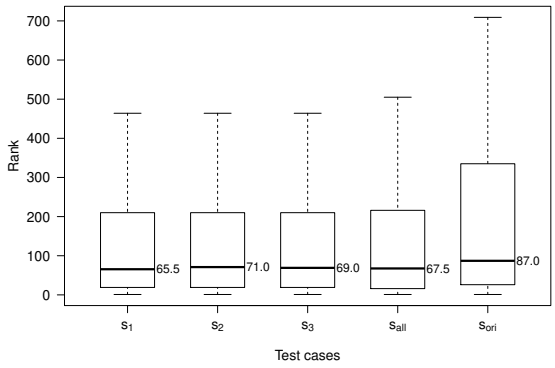

Figure 4: Comparison of the rank of the faulty statements

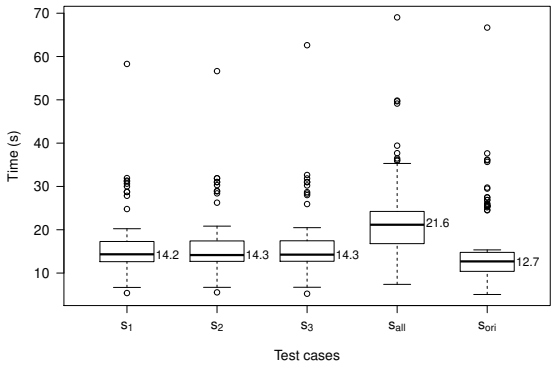

Figure 5: Comparison of the execution time

the same suspiciousness, the rank of the faulty one is five. This metric means how many statements developers need to check until they identify the faulty statements.

4.3 Analysis for the accuracy (RQ1)

In 106 Math bugs, there were 220 faulty statements for which were calculated the suspiciousness. We compared the rank of the faulty statements for the test cases S_1, S_2, S_3, S_{all}, and S_{ori}. Table 1 shows the number of faulty statements where adding test cases improves the accuracy of FL, has no impact, and worsens, compared with S_{ori}. S_1 improved the accuracy for 56.3% faulty statements while it worsened the accuracy for 17.3% ones, which was the best result in S_1, S_2, and S_3. From those results, the same-spectrum test cases with different pass/fail statuses and failed test cases may not correct false negative results but worsens the results. Comparing the results of S_{all} and S_1, S_{all} more improved the accuracy than S_1 by 30 faulty statements, but the number of the faulty statements worsened was the same. In both cases, the accuracy of FL was improved or the same for 82.7% of the 220 faulty statements.

Figure 4 shows a box plot of the rank of the faulty statements[3]. The vertical axis presents the rank of faulty statements, and lower values mean higher accuracy. The results show adding test cases improved the accuracy of FL. The medians of the rank of the faulty statements when S_1, S_2 S_3, S_{all} or S_{ori} are used are 65.5, 71.0, 69.0, 67.5 or 87.0, respectively. There was little difference due to the strategy for adding test cases, but S_1 most improved the accuracy. The rank of the faulty statements was improved by 21.5 at the median when S_1 was used.

4.4 Analysis for the execution time (RQ2)

Figure 5 shows a box plot of execution time. The vertical axis presents the execution time. The results show the execution time is reduced when test cases are selected, compared with S_{all} was used. The medians of the execution time when S_1, S_2, S_3, S_{all}, or S_{ori} are used are 14.2s, 14.3s, 14.3s, 21.6s, or 12.7s, respectively. The increase in the execution time due to the addition of selected test cases was suppressed to the median of 1.5 seconds, which was 7.4s shorter than when all the S_{all} was used.

Table 1: The number of faulty statements where adding test cases improves the accuracy of FL (column Positive), has no impact (column Neutral), and worsens (column Negative).

Test cases	Positive	Neutral	Negative
S_1	124 (56.3%)	58 (26.4%)	38 (17.3%)
S_2	122 (55.5%)	58 (26.4%)	40 (18.1%)
S_3	122 (55.5%)	47 (21.3%)	51 (23.2%)
S_{all}	154 (70.0%)	28 (12.7%)	38 (17.3%)

[3]Outliers are hidden for visibility

5 THREATS TO VALIDITY

In our experiment, we used EvoSuite to generate test cases. EvoSuite uses a randomized algorithm to generate test cases. Consequently, it is possible to get different results if the same experiment under the same conditions is conducted. We only used Math in Defects4J as the experimental subject, although there are many datasets of bugs. It is possible to get different results if subject programs with different characteristics are used for the experiment.

6 CONCLUSION

In this paper, we proposed a technique to obtain test cases having different spectra from original test cases in order to increase the accuracy of SBFL while suppressing the increase in execution time. In the experiment, the test cases obtained by the proposed technique were added to the original test cases, and then SBFL was performed. The accuracy of SBFL for 82.7% of the faulty statements for the target defects is the same as or better than the SBFL using only the original test cases. The increase in execution time due to the addition of test cases by the proposed technique was suppressed to a median of 1.5 seconds, which was 7.4 seconds shorter than when all the generated test cases were used.

As the next step of this research, we are going to incorporate the proposed technique into APR. The experiment in this paper shows the possibility that using the test cases generated by the proposed technique in addition to the original ones improves the accuracy of FL with a slight increase in execution time. However, even if the increase in the execution time required for FL at a single time is slight, it can have a significant effect when performing FL at multiple times in APR. Thus, it is necessary to investigate how the test cases added by the proposed technique affect the results and the execution time of APR.

ACKNOWLEDGMENT

This work was supported by MEXT/JSPS KAKENHI 17H01725.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[2] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5 (2014), 507–525.

[3] Andréia da Silva, Meyer, Antonio Augusto, Franco Garcia, Anete Pereira, de Souza, and Cláudio Lopes de Souza, Jr. 2004. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L). *Genetics and Molecular Biology* 27, 1 (2004), 83–91.

[4] Gong Dandan, Wang Tiantian, Su Xiaohong, and Ma Peijun. 2013. A test-suite reduction approach to improving fault-localization effectiveness. *Computer Languages, Systems & Structures* 39, 3 (2013), 95–108.

[5] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 416–419.

[6] Wei Jin and Alessandro Orso. 2013. F3: Fault Localization for Field Failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 213–223.

[7] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 437–440.

[8] B. Korel. 1988. PELAS-program error-locating assistant system. *IEEE Transactions on Software Engineering* 14, 9 (1988), 1253–1260.

[9] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2019. The Impact of Rare Failures on Statistical Fault Localization: the Case of the Defects4J Suite. In *2019 IEEE International Conference on Software Maintenance and Evolution*. 24–28.

[10] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2011. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering* 38, 1 (2011), 54–72.

[11] Ning Li, Rui Wang, Yu-li Tian, and Wei Zheng. 2016. An Effective Strategy to Build Up a Balanced Test Suite for Spectrum-Based Fault Localization. *Mathematical Problems in Engineering* 2016 (2016), 1–13.

[12] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. 2005. SOBER: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 286–295.

[13] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 273–283.

[14] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.