IEICE TRANS. INF. & SYST., VOL.E97–D, NO.6 JUNE 2014

LETTER

Enriching Contextual Information for Fault Localization

Zhuo ZHANG[†a)], **Xiaoguang MAO**[†b)], *Nonmembers*, **Yan LEI**[†], *Student Member*, *and* **Peng ZHANG**[†], *Nonmember*

SUMMARY Existing fault localization approaches usually do not provide a context for developers to understand the problem. Thus, this paper proposes a novel approach using the dynamic backward slicing technique to enrich contexts for existing approaches. Our empirical results show that our approach significantly outperforms five state-of-the-art fault localization techniques.

key words: *fault localization, dynamic slice, program spectrum, contextual information*

1. Introduction

Since debugging is generally recognized as an expensive and time-consuming process, researchers have proposed many useful techniques to provide assistance in finding the faults that cause programs to produce incorrect outputs. However, they just focus on statement selection and ranking, and ignore the contextual information of the relationship and the propagation among suspicious statements [1]. Hence, it is necessary to construct the contextual information that can be helpful in improving the activity of debugging.

Recently, spectrum-based fault localization (SFL) [2] is a popular topic of study and used in the fault localization community. However, SFL just outputs a ranked list of isolated entities and fails to provide the contextual information for discovering and understanding how suspicious statements act on each other to trigger failure. Moreover, SFL uses code coverage that cannot distinguish whether the execution of a statement affected the output [4]. However, the execution of faulty statements should generally trigger and affect the faulty output of a program when a failure occurs. Due to the use of coverage information, SFL may include additional statements unrelated to a failure. For example, SFL will assign high values of suspicion to those statements that are frequently executed by failed test cases even if those statements do not affect the faulty outputs of the program. To address these issues, we propose an approach that leverages program slicing to enrich the contextual information and reduce the searching scope of suspicious statements for SFL.

Program slicing techniques utilize data and control dependencies to identify a set of statements that may affect the value at a specific statement of a program [4], [6]. The set of statements is denoted as a program slice. A program slice is essentially a context illustrating a causal chain of how data and control propagate in a program. Experiments have shown that a slice is useful for understanding and discovering faults [10]. Nevertheless, program slicing techniques cannot distinguish which statement in a slice is more suspicious and thus should be examined and so developers are frustrated and tired as slice size is usually large to match the increasing levels of complexity of today's software [1]. It is necessary to give some examining guidance for further reducing the heavy burdens placed on the developers.

Thus, this paper proposes an effective approach to enrich contexts for a promising fault localization technique (this is SFL) and provide examining guidance for developers. Our approach uses dynamic slicing techniques to slice the faulty output of a failed test case to construct a dynamic slice. Since the dynamic slice shows that how the faulty output is produced, our approach identifies this dynamic slice as a suspicious context. Then, our approach adopts SFL to quantify the suspiciousness of the statements in the context being faulty to recommend examining guidance for developers. Consequently, our approach provides useful contexts and recommends examining guidance of their elements in terms of suspiciousness, that is, our approach enriches the contexts for fault localization techniques. We conduct an empirical study on 6 representative programs in fault localization with 61 faulty versions. The results show that our approach can reduce almost 16.56% up to 67.05% of the average cost of examined code over five maximal SFL formulas, namely ER1',ER5, GP02,GP03 and GP19.

2. Our Approach

2.1 Overview

Program slicing as a debugging aid was introduced by Mark Weiser [8]. Later Korel and Laski proposed dynamic slicing to focus on an execution in a specific input [9]. Compared to the static slicing technology which analyzes programs without running them, the dynamic slicing technology gathers run-time information along the execution path and noticeably cuts down the size of the slice in comparison with static slicing. The basic idea of our approach is to use dynamic backward slicing to slice the failed output to construct a suspicious slice as a suspicious context, and then adopt SFL to quantify the suspiciousness of the statements in the suspi-

Manuscript received January 10, 2014.
Manuscript revised March 1, 2014.

[†]The authors are with College of Computer, National University of Defense Technology, 410073 Changsha, China.
a) E-mail: zhuozhangnudt@gmail.com
b) E-mail: xgmao@nudt.edu.cn

DOI: 10.1587/transinf.E97.D.1652

$$\begin{matrix} & N\,statements & & errors \\ \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix} & & & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \end{matrix}$$

Fig. 1 The coverage of M executions.

cious context, and finally present the suspicious context and its statements with different suspiciousness for developers. Since a dynamic slice can show useful data/control dependencies of how statements act on each other, the suspicious context defined in our approach can show more useful information for developers to understand and discover the fault. In addition, our approach assigns different suspiciousness of a statement being faulty and therefore it is practical for developers to quickly distinguish which statements are more suspicious and should be firstly checked.

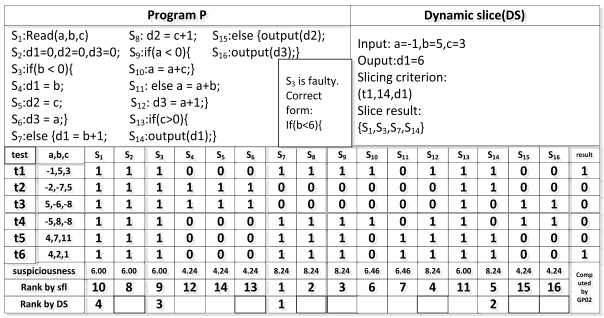
Step 1: Construct suspicious context and their elements with dynamic slicing techniques. In this step, we compute the dynamic slices [9] of the incorrect output of a failed test case because this dynamic slice can show how data/control flow propagates and finally results in the failure of the failed test case. We identify this dynamic slice as a suspicious context that contains a set of suspicious statements and demonstrates how each statement acts on each other to cause a failure.

Step 2: Compute the suspiciousness of each statement in the context. In this step, we adopt SFL [2] to compute the suspiciousness of each statement in the context. SFL utilizes the statement coverage data which is collected indicating that whether the statements are executed or not during execution of test cases to calculate the suspiciousness of a statement. Given a program P, it consists of N executable statements and is executed by M test cases T that contains at least one failed test case (see Fig. 1). $x_{ij} = 1$ represents that statement j is executed under test case i and $x_{ij} = 0$ otherwise. The error vector e means the test results. The element e_i is equal to 1 if test case i failed, and 0 otherwise. Based on the matrix $M \times (N + 1)$, four parameters a_{np}, a_{nf}, a_{ep}, a_{ef} are defined for each statement showing the number of passed/failed test cases in which the statement was/wasn't executed. The first part of the subscript indicates whether the statement was executed (e) or not (n) while the second suggests whether the test case passed (p) or failed (f), four parameters a_{np}, a_{nf}, a_{ep}, a_{ef} are defined for each statement showing the number of passed/failed test cases in which the statement was/wasn't executed. The first part of the subscript indicates whether the statement was executed (e) or not (n) while the second suggests whether the test passed (p) or failed (f).

Based on the above definitions, we can use the formulas listed in Table 1 to calculate the suspiciousness of each statement in a suspicious context. Xie et al. [2], [7] have conducted a theoretical analysis of 60 risk evaluation formulas and found 9 the most efficient formulas (referred to as the maximal formulas). In light of equivalence, these formulas form five groups, namely ER1', ER5, GP02, GP03 and GP19 (See Table 1). After suspiciousness calculation, a ranking list of all statements in the context is produced in descending order of their suspiciousness.

Table 1 Formulas of SFL.

Name		Formulas
ER1'	Naish1	$\begin{cases} -1 & \text{if } a_{ne} > 0 \\ a_{np} & \text{if } a_{ne} \le 0 \end{cases}$
	Naish2	$a_{ef} - \frac{a_{ep}}{a_{ep} + a_{np} + 1}$
	GP13	$a_{ef}\left(1 + \frac{a_{ep}}{2a_{ep} + a_{ef}}\right)$
ER5	Wong1	a_{ef}
	Russel&Rao	$\frac{a_{ef}}{a_{ef} + a_{nf} + a_{ep} + a_{np}}$
	Binary	$\begin{cases} 0 \text{ if } a_{ne} > 0 \\ 1 \text{ if } a_{ne} \le 0 \end{cases}$
GP02		$2\left(a_{ef} + \sqrt{a_{np}}\right) + \sqrt{a_{ep}}$
GP03		$\sqrt{\lvert a_{ef}^2 - \sqrt{a_{ep}} \rvert}$
GP19		$a_{ef}\sqrt{\lvert a_{ep} - a_{ef} + a_{nf} - a_{np} \rvert}$



test	a,b,c	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}	S_{13}	S_{14}	S_{15}	S_{16}	result
t1	-1,5,3	1	1	1	0	0	0	1	1	1	1	0	1	1	1	0	0	1
t2	-2,-7,5	1	1	1	1	1	1	0	0	0	0	0	0	1	1	0	0	0
t3	5,-6,-8	1	1	1	1	1	1	0	0	0	0	0	0	1	0	1	1	0
t4	-5,8,-8	1	1	1	0	0	0	1	1	1	1	0	1	1	0	1	1	0
t5	4,7,11	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	0
t6	4,2,1	1	1	1	0	0	0	1	1	1	0	1	1	1	1	0	0	1
suspiciousness		6.00	6.00	6.00	4.24	4.24	4.24	8.24	8.24	8.24	6.46	6.46	8.24	6.00	8.24	4.24	4.24	Computed by GP02
Rank by sfl		10	8	9	12	14	13	1	2	3	6	7	4	11	5	15	16	
Rank by DS		4		3				1							2			

Fig. 2 Example illustrating the context-enriching approach.

2.2 An Illustrative Example

Figure 2 illustrates a faulty program P that contains a faulty statement s_3 to show just how our approach is to be applied. This example chooses one type of SFL, GP02, to compute the suspiciousness of each statement. The cells below each statement indicate whether the statement is covered by the execution of a test case or not (1 for executed and 0 for non-executed) and the rightmost cells represent whether the execution of a test case is failed or not (0 for pass and 1 for fail). Based on the statement coverage information and test results of six test cases, GP02 outputs a ranking list of all statements in descending order of suspiciousness: {s_7, s_8, s_9, s_{12}, s_{14}, s_{10}, s_{11}, s_2, s_3, s_1, s_{13}, s_4, s_6, s_5, s_{15}, s_{16}}, but this set includes all statements and cannot distinguish which statement affects the output. For example, s_8 and the faulty statement s_3 are executed by all failed test cases. However, s_8 is less frequently to be executed by passed test cases compared with s_3. In this case, s_8 has a higher suspicious value than s_3 even if s_8 does not affect the faulty outputs. It reveals the reason that the faulty statement s_3 is ranked 9th while

some statements unrelated to failures are ranked higher than s_3.

To address this issue, our approach selects a failed test case t_1 and constructs a dynamic slice of the failed output of t_1 by using a slicing criterion (t_1, s_{14}, d_1). This dynamic slice $\{s_1, s_3, s_7, s_{14}\}$ constructs a suspicious context that contains statements are responsible for the failed output of t_1. With the dynamic slice, GP02 generates a new ranking list: $\{s_7, s_{14}, s_3, s_1\}$, s_3 is ranked 3rd. $\{s_2, s_4, s_5, s_6, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{15}, s_{16}\}$ are not included in the context because they do not affect the failed output of t_1. Therefore, our approach obtains a better localization result than original GP02.

3. An Experimental Study

3.1 Experiment Setup

The experiment selects six Java programs. Table 2 lists the name of these programs, function description, the number of faulty versions, the lines of code, the executable lines of code and the number of test cases. The all six subject programs are Java version of Siemens programs that are widely used in the fault localization community.

Since the research [2] has theoretically proven that 9 out of 60 SFL suspiciousness evaluation formulas are the most efficient formulas (referred to as the maximal formulas), we choose these formulas in our study. We evaluate SFL by only using Nash 1 out of the three equivalent formulas in ER1', Binary out of the three equivalent maximal formulas in ER5 and the other three formulas GP02, GP03 and GP19. In light of the equivalence in each group, the following section uses ER1' and ER5 to represent Naish1 and Binary respectively [2].

3.2 Data Analysis

We utilize a widely used metric *EXAM* [2] that is defined as the percentage of executable statements to be examined before finding the actual faulty statement. For a more detailed comparison, we further adopt *Imp* [3] that is to compare the total number of statements that need to be examined to find all faults by our approach versus the number that need to be examined by using the SFL. As a reminder, our system uses EMMA [5] to collect code coverage and JSlice [6] to perform dynamic slicing.

Figure 3 illustrates the *EAXM* score of our approach over its corresponding SFL approaches in all faulty versions. As shown in Fig. 3, we observe that our approach obtains a significant improvement over ER1', GP02 and GP13 when the *EXAM* begins at 5%. Additionally, our approach gets an obvious improvement over ER5 and GP19 starting at the *EXAM* of 15% and 10% respectively. Therefore, our approach significantly improves the effectiveness of all five maximal formulas.

Figure 4 shows the *Imp* score of our approach over each SFL formula in each program. With our approach, the statements that need to be examined sharply decrease, ranging from 32.95% such as GP02 on schedule2 to 83.44% such as ER1' on Jtcas. This means that we only need to examine from 32.95% to 83.44% of executed statements that SFL needs to examine of. It also represents that our approach obtains the saving from 67.05% (100% − 32.95% = 67.05%) to 16.56% (100% − 83.44% = 16.56%) over SFL in terms of effort. As shown in Fig. 5, the maximum saving is 67.05% on GP02 in schedule2 while the minimum saving is 16.56% on ER1' in Jtcas, the average saving ranges from 40.00% to 54.25%. In conclusion, our approach is effective to improve fault localization effectiveness.

Table 2 The characteristics of subject programs.

Program	Description	Versions	LOC	Ex	Test
Print tokens1	lexical analyzer	5	587	210	4071
Print tokens2	lexical analyzer	10	571	223	4056
Schedule1	priority scheduler	9	422	151	2650
Schedule2	priority scheduler	10	383	153	2710
Tot info	Information measure	23	411	153	1052
Jtcas	collision avoidance	14	198	87	1608

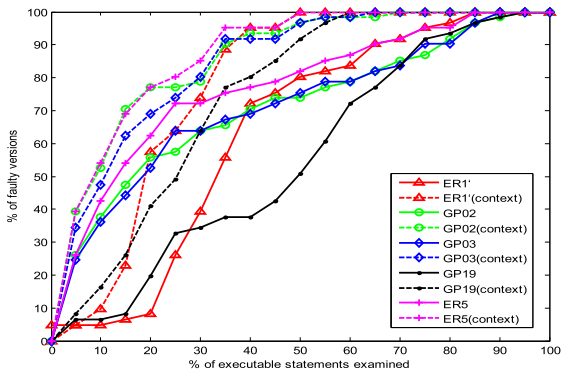


Fig. 3 *EXAM* comparison between SFL and our approach.

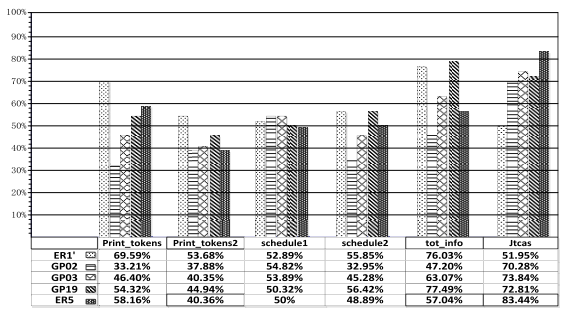


Fig. 4 *Imp* of our approach.

3.3 Threats to Validity

First and foremost, chances are that fault statement may not be included in the slice result such as version 11 of Tot info. That drawback is caused by characteristic of dynamic slicing technology. Even though, a wide spectrum type of faults can be included by dynamic slicing technology [10]. An-

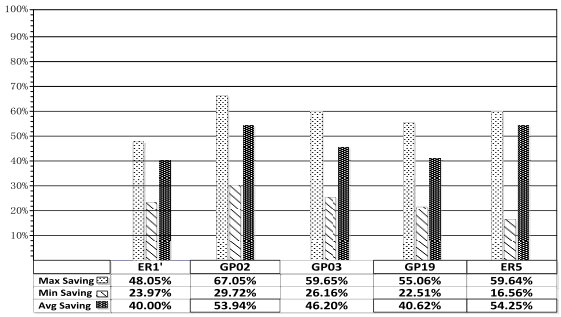


Fig. 5 *Saving* of our approach over SFL.

other threat is the subject programs used by our experiment. The subject programs are all small-sized programs despite of their widely application in the fault localization community. It is worthwhile to conduct more experiments on large-scale programs.

4. Conclusion

In this paper, we propose an approach using dynamic slicing to enrich contexts for fault localization and the results show a preliminary benefit on fault localization. In the future, we plan to use more large-scale Java programs and conduct user studies. Moreover, we will consider an extension of our current work to multiple-bug cases.

Acknowledgements

This work is partially supported by the National Natural Science Foundation of China under Grant No.61379054 and No.91118007, the National High Technology Research and Development Program of China (863 program) under Grant No.2012AA011201.

References

[1] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?," ISSTA 2011, pp.199–209, 2011.
[2] X. Xie, F.-C. Kuo, T.Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in SBSE for spectrum based fault localisation," SSBSE 2013, pp.224–238, 2013.
[3] V. Debroy, W.E. Wong, X. Xu, and B. Choi, "A grouping-based strategy to improve the effectiveness of fault localization techniques," QISC 2010, pp.13–22, 2010.
[4] Y. Lei, X. Mao, Z. Dai, and C. Wang, "Effective statistical fault localization using program slices," COMPSAC 2012, pp.1–10, 2012.
[5] EMMA, http://emma.sourceforge.net/
[6] JSlice, http://jslice.sourceforge.net/
[7] 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 Trans. Software Engineering and Methodology, pp.1–39, 2013.
[8] M. Weiser, "Program slicing," IEEE Trans. Softw. Eng., vol.10, pp.352–357, 1984.
[9] B. Korel and J. Laski, "Dynamic program slicing," Information Processing Letters, vol.29, pp.155–163, 1988.
[10] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," Empirical Software Engineering, vol.12, pp.143–160, 2007.