IEICE TRANSACTIONS

on Information and Systems

VOL. E98-D NO. 4
APRIL 2015

A PUBLICATION OF THE INFORMATION AND SYSTEMS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3 chome, Minato-ku, TOKYO, 105-0011 JAPAN

LETTER

Fault Localization Using Failure-Related Contexts for Automatic Program Repair

Ang LI[†a)], ***Student Member***, **Xiaoguang MAO**[†,††b)], ***Nonmember***, **Yan LEI**[†], ***Student Member***, ***and*** **Tao JI**[†], ***Nonmember***

SUMMARY Fault localization is essential for conducting effective program repair. However, preliminary studies have shown that existing fault localization approaches do not take the requirements of automatic repair into account, and therefore restrict the repair performance. To address this issue, this paper presents the first study on designing fault localization approaches for automatic program repair, that is, we propose a fault localization approach using failure-related contexts in order to improve automatic program repair. The proposed approach first utilizes program slicing technique to construct a failure-related context, then evaluates the suspiciousness of each element in this context, and finally transfers the result of evaluation to automatic program repair techniques for performing repair on faulty programs. The experimental results demonstrate that the proposed approach is effective to improve automatic repair performance.

key words: *automatic repair, fault localization, failure-related context, program slicing, suspiciousness evaluation*

1. Introduction

Fixing software bugs is a painstaking task in software development and maintenance [1]. According to the recent research, the cost related with software testing, debugging and fixing has accounted for around 50% in an entire project, sometimes even reaching 90% [2], [3]. In order to reduce the cost of fixing bugs, researchers have devoted themselves to realizing the automation of software repair, and proposed many automatic repair techniques in recent years, for instance, JAFF [4], [5], Par [6], etc. Among all of these techniques, the most notable and influential one is GenProg, proposed by University of Virginia [7], [8]. Those researchers extend the conception of genetic algorithm, and put it into use in automatic repair, and develop GenProg, which is an effective automatic repair tool for C language. The large-scale experiments show that GenProg succeeds in fixing 55 bugs out of 105 from those typical open-source software such as libtiff, php, python, wireshark, etc [7]. Although it indicates that GenProg performs well in automatic repair and embodies great practical value, it is still not satisfactory in both effectiveness and efficiency, and needs to be further improved. For example, GenProg takes the powerful Amazon EC2 cloud computing platform to execute its benchmarks, but it consumes a large amount of resources, with every successful repair taking 96 minutes by average [7].

Automatic repair generally consists of three parts: fault localization, patch generation and patch verification. Being the first part, fault localization mainly provides the information of possible faulty locations, and the follow-up steps will conduct the mutation operations on these possible faulty locations to generate a valid patch as fast as possible. Thus, the accuracy of fault localization greatly impacts the performance of the follow-up steps and the whole repair process. According to our previous research [9], developers mainly design and evaluate the fault localization module from the view of manual fixing, without considering the requirements of automatic repair. We have found that the current fault localization approaches focus on how to obtain a high rank for the faulty statements in the set of suspicious statements. These approaches can attract developers' attention to these high-rank statements, and thus help them to repair the program. However, these techniques just output the isolated ranked statements and does not provide any contextual information. Since developers have the knowledge of the program, they can construct the context by themselves to understand the relationship among these isolated ranked statements and finally pinpoint the location of the fault.

However, automatic repair lacks the knowledge as developers have, and thus cannot construct a context by itself to speed up the search of the location of the fault. In this case, when obtaining fault localization results, existing automatic repair techniques generate a valid patch merely from the perspective of the whole program. Since the whole program has many statements which may have nothing to do with a failure, if we conduct the automatic repair starting from the view of all the program codes, it will produce a large number of invalid patches until we find a valid patch. Therefore, this can lead to a high cost in automatic repair, and greatly restrict the performance of patch generation and patch verification, and finally degrade the entire repair performance. For improving the whole performance of automatic program repair, it is necessary to study the fault localization from the view of automatic repair.

To the best of our knowledge, there is no study aiming at developing effective fault localization approaches for automatic repair. This paper tries to fill this void and propose a fault localization approach using failure-related con-

Manuscript received September 19, 2014.
Manuscript revised December 4, 2014.
Manuscript publicized January 8, 2015.

[†] The authors are with College of Computer, National University of Defense Technology, 410073, Changsha, China.

[††] The author is also with Laboratory of Science and Technology on Integrated Logistics Support, National University of Defense Technology, Changsha 410073, China.

a) E-mail: liooooon@hotmail.com

b) E-mail: xgmao@nudt.edu.cn (Corresponding author)

DOI: 10.1587/transinf.2014EDL8189

texts to improve the performance of automatic repair. This approach puts all the suspicious statements into a failure-related context, which can provide useful guidance information for patch generation and finally improve the whole repair performance. There are mainly three phases in this approach, and firstly we try to construct the failure-related context from several failed test cases' outputs. Our previous work on constructing a context [10] inspires us to use program slicing technology to realize it. Secondly, we apply Spectrum-based Fault Localization (SFL) [11] to implement suspicious value evaluation of the elements in the context. Finally, we present the evaluation result to the follow-up automatic repair module to finish the repair work. Considering the popularity and influence of GenProg, we apply our approach on it, that is, we replace GenProg's original fault localization module with ours. The experimental results show that our approach can effectively reduce the number of invalid patches generated, significantly reduce the time cost, and thus improve the whole performance of automatic repair.

2. Fault Localization Using Failure-Related Contexts (FLFC)

M. Weiser first proposed program slicing[†] to assist researchers in debugging programs [12]. Since M. Weiser's approach is static program slicing, it considers all data and control dependency in all possible inputs and thus obtains relatively large slices. For reducing the size of those slices, B. Korel and J. Laski [13] proposed dynamic program slicing by considering all data and control dependency in a specific input. By using the data and control dependency, researchers use program slicing technology to identify those statements that have direct or indirect data and control dependency with a failure, and those statements are denoted as a slice. More specifically, when a failure happens in a program, its faulty output should be related to the fault. Therefore, we conduct backward slicing on the faulty output to obtain a slice. The statements of this slice have data and control dependency on the computation value of the faulty output. The previous studies [10], [14], [15] have shown that this slice can effectively include the faulty statements and some critical statements relevant with the faults. It means that if automatic repair conduct mutation operations on the elements of this slice to generate patch candidates, it may have a higher probability to obtain a valid patch among the patch candidates and thus reduce the number of invalid patches and the whole repair time. Based on the analysis above, we propose an approach called Fault Localization using Failure-related Contexts (FLFC), to construct failure-related contexts and thus give more guidance on the follow-up patch generation and verification. The main idea of FLFC is to implement SFL on the statements appearing in the failure-related context to get the fault localization result. So our goal is to replace the original fault localization module of GenProg with FLFC. The following content will describe the process of FLFC.

Step 1: Construct a failure-related context. A failure-related context is defined by backward slicing. First, a backward slice is usually defined by two factors: a statement s in the program and a variable v at the statement s. Give a slicing criterion (s, v), backward slicing will output a set of statements (denoted as a slice) that affect the computation of the value of variable v at statement s. Suppose that we have a failed test case t_f, and the variable v_f at the statement s_f outputs the faulty value that mismatches the value in the test oracle. A failure-related context for the failure of t_f is a backward slice against the faulty output of the failed test case t_f, that is, it is the backward slice in case of the slicing criterion (v_f, s_f). Thus, a failure-related context for t_f can identify those statements affecting the faulty value that leads the program to have a failure. For automatic repair, the follow-up patch generation will apply mutation operations on those statements to generate patch candidates. Since the statements of a failure-related context are highly related to a failure, it means that we have a higher probability to obtain a valid patch among the patch candidates and thus improve the performance of automatic repair. For a given bug program, our approach will select several critical failed test cases to construct a failure-related context for the follow-up repair activities.

To make our idea more explicit, we give a simple example below:

```
1  int main(int argc, char ** argv)
2  {
3  int number[10];
4  int a=0,b=1,c=2,d=3;
5  while(a<20)
6  {
7     a+=b;
8     //an array boundary exception
9     //happens when a equals to 10
10    number[a]=a;
11    d=d-b;
12    c=c+d;
13    //faulty output when a equals to 10
14    printf("%d",number[a]);
15 }
16
17 return 0;
18
19 }
```

It is evident that an array boundary exception of the array "number" happens when the variable "*a*" equals to 10, and line 14 is a output statement relevant to the array "number". Therefore, we conduct backward slicing with a slicing criterion (line: 14, varible: number) to obtain a backward slice that can identify those statements affecting the computation value of the variable at the line 14. This slice is the failure-related context including the lines of 3, 4, 5, 7, 10 and 14. As shown in this example, we successfully exclude those lines which have no relationship with this exception.

Step 2: Evaluate each element's suspiciousness value in

[†] In this paper, program slicing is backward slicing and they are used interchangeably.

the failure-related context. Furthermore, it is necessary to identify the suspiciousness of being faulty for each statement in the failure-related context on the program output and give more guidance to patch generation. Therefore, this step adopts SFL to evaluate the suspiciousness value for the statements which only appear in the failure-related context, and thus uses the suspiciousness value to measure the magnitude of the correlation of each statement with the program behavior (that is program output). If a statement has a high suspiciousness value, it means that this statement has stronger influence on the output. Therefore, the automatic repair program will choose and mutate this statement with a high priority in follow-up patch generation to generate patch candidates.

In contrast, the original fault localization module of GenProg uses SFL to evaluate the suspiciousness value of all statements in the program. Therefore, all the statements are potential to be mutated, and a statement with a high suspiciousness value will have higher priority to be chosen and mutated in follow-up patch generation to generate patch candidates. Since the number of all statements is too large, GenProg will perform more mutation operations on more statements to obtain a valid patch, and thus this would degrade the repair performance. Actually, many statements have no relation with the program failure so it is unnecessary to inspect all of them. If we can exclude those irrelevant statements and only evaluate the suspiciousness value of those relevant statements instead of all statements of the program, we can effectively narrow down the scope of the statements to be mutated, and thus obtain a valid patch with fewer trials. Hence, we conduct SFL on the statements only appearing in the failure-related context from step 1 instead of all statements. Since the failure-related context can exclude those statements that have no influence on the program output, it will highly improve the searching process of a valid patch in the current methodology of program repair. The algorithm of SFL on the failure-related context is described as follows:

Firstly, suppose that there is a program P, with its failure-related context $S = \{s_1, s_2, \ldots, s_M\}$ executed in the test case suite $T = t_1, t_2, \ldots, t_N$, where $M = |S|$, $N = |T|$. As shown in Fig. 1, the $N \times (M + 1)$ matrix is the input of SFL. An element x_{ij} is 1 if the statement s_j is covered by the execution of test case t_i, 0 otherwise. The result vector r at the rightmost column of the matrix represents the test results. The element r_i is 1 if t_i is failed, 0 otherwise. Based on the matrix defined in Fig. 1, four statistical variables are defined: $a_{00}(s_j)$ and $a_{01}(s_j)$ represent the numbers of test cases that do *not* execute the statement s_j, and return the *passing* and *failing* test results, respectively; $a_{10}(s_j)$ and $a_{11}(s_j)$ stand for the numbers of test cases that *execute* s_j, and return the *passing* and *failing* testing results, respectively. With the four statistical variables, many suspiciousness evaluation formulas are proposed for SFL. Eq. 1 shows how Jaccard formula computes the suspiciousness value of statement s_j.

$$\begin{array}{ccc}
 & \textit{M statements} & \textit{failed/passed} \\
\textit{N test cases} & \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1M} \\ x_{21} & x_{22} & \cdots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{NM} \end{bmatrix} & \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_N \end{bmatrix} \begin{matrix} t_1 \\ t_2 \\ \vdots \\ t_N \end{matrix} \\
 & \begin{matrix} s_1 & s_2 & \cdots & s_M \end{matrix} &
\end{array}$$

Fig. 1 Input of SFL.

$$Jaccard(s_j) = \frac{a_{11}(s_j)}{a_{11}(s_j) + a_{01}(s_j) + a_{10}(s_j)} \tag{1}$$

Take the above simple program as an example. FLFC will only evaluate the suspiciousness values of the lines of 3, 4, 5, 7, 10 and 14, and the other lines are excluded. On the contrary, the original GenProg will evaluate the suspiciousness values of all statements.

Step 3: Input the result of FLFC into the patches generation to conduct automatic repair. This step input the result of FLFC into the patches generation stage, that is, we present the statements in the failure-related context and their suspiciousness values to the follow-up process. The patch generation module will apply mutation operations only on the statements appearing in the failure-related context. It will first choose and mutate those statements with a high suspiciousness value to generate patch candidates. In contrast, the original GenProg will evaluates the suspiciousness values for all statements, which means those statements irrelevant to the faulty output also have the chance to be chosen and mutated in the follow-up process.

3. Experimental Study

3.1 Subject Programs and Evaluation Metrics

To evaluate our approach, the experiments selected the subject C programs used in the most recent work [7] on GenProg as the experimental benchmarks. Since the automatic repair needs a lot of computational resources, the current experiments include 10 different bugs. Table 1 lists the programs, lines of code, the number of test cases and the bug version. We prepare two different kinds of fault localization modules: our fault localization module FLFC and the original one of GenProg. The first module is FLFC. Since FLFC

Table 1 Subject programs.

Program	LOC	Test Cases	Version
libtiff	77,000	59	bug-6f9f4d7-73757f3
		31	bug-0860361d-1ba75257
		33	bug-10a4985-5362170
		73	bug-0fb6cf7-b4158fa
		73	bug-01209c9-aaf9eb3
		64	bug-5b02179-3dfb33b
		73	bug-d39db2b-4cd598c
python	407,000	303	bug-69783-69784
php	1,046,000	4,986	bug-309892-309910
wireshark	2,814,000	53	bug-37112-37111

needs a SFL formula and the study [9] has shown that one of many SFL formulas called Jaccard is the best suspiciousness evaluation formula for automatic repair, we choose Jaccard formula to perform suspiciousness evaluation. FLFC will provide the statements only in the failure-related context and their suspiciousness values for the follow-up patch generation process. The second module is the original fault localization module of GenProg: a single Jaccard module without failure-related contexts. This module will present all statements of the program and their suspiciousness values to the follow-up patch generation process. We will compare their performance in program repair using the benchmark programs. The Jaccard formula is shown in Eq. 1.

The experiments evaluate the repair performance from two aspects, namely effectiveness and efficiency. For effectiveness, the experiments adopt Number of Candidate Patches (NCP) [9]: the number of invalid patches generated until a valid patch is found. A smaller NCP indicates better effectiveness. For efficiency, we take the time consumed during the whole repair process as the criterion. Lower time cost represents higher efficiency.

All the experiments ran on an Ubuntu 10.04 machine with 2.35 GHz Intel quad-core CPU and 2 GB of memory. For each version of bugs, we ran the two different automatic repair approaches, equipped with FLFC and the original single Jaccard module respectively, 20 times to obtain our experiment results. We choose *Unravel* and *Gcov* as our slicing tools.

3.2 Results and Analysis

In order to compare the performance between FLFC and the single Jaccard module, the study will obtain the repair time and NCP with 10 different bugs, and analyze the experiment results from two aspects: the boxplots and the paired Wilcoxon-Signed-Rank Test.

Figure 2 and Fig. 3 use the boxplots to show the repair time and NCP of FLFC over Jaccard. As shown in the two figures, the repair time and NCP of the GenProg

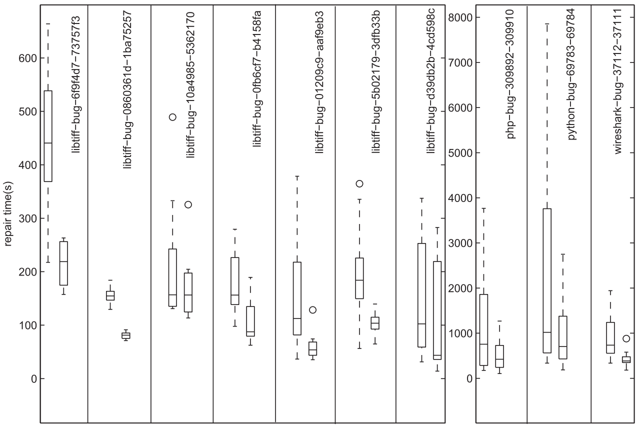

Fig. 2 Boxplots on the repair time.

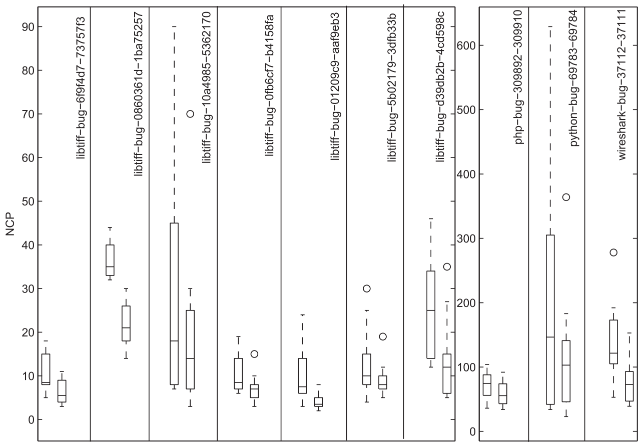

Fig. 3 Boxplots on the NCP.

Table 2 Wilcoxon-signed-rank test on repair time and NCP.

Program		2-tailed	1-tailed (right)	1-tailed (left)	Conclusion
libtiff-bug-6f9f4d7-73757f3	Repair time	5.062032e-03	9.978415e-01	2.960769e-03	BETTER
	NCP	7.801281e-03	9.966532e-01	4.535230e-03	BETTER
libtiff-bug-0860361d-1ba75257	Repair time	5.062032e-03	9.978415e-01	2.960769e-03	BETTER
	NCP	4.670906e-03	9.980137e-01	2.739277e-03	BETTER
libtiff-bug-10a4985-5362170	Repair time	2.411214e-01	8.893641e-01	1.310963e-01	BETTER
	NCP	2.135244e-01	9.037412e-01	1.180685e-01	BETTER
libtiff-bug-0fb6cf7-b4158fa	Repair time	5.062032e-03	9.978415e-01	2.960769e-03	BETTER
	NCP	1.192100e-02	9.949785e-01	7.051958e-03	BETTER
libtiff-bug-01209c9-aaf9eb3	Repair time	9.344113e-03	9.959774e-01	5.413461e-03	BETTER
	NCP	7.922776e-03	9.965992e-01	4.603390e-03	BETTER
libtiff-bug-5b02179-3dfb33b	Repair time	2.841686e-02	9.875338e-01	1.615646e-02	BETTER
	NCP	2.094824e-01	9.057033e-01	1.159989e-01	BETTER
libtiff-bug-d39db2b-4cd598c	Repair time	2.845027e-01	8.689037e-01	1.540316e-01	BETTER
	NCP	5.500973e-01	7.445303e-01	2.953393e-01	BETTER
php-bug-309892-309910	Repair time	9.260070e-02	9.584344e-01	5.145877e-02	BETTER
	NCP	1.394140e-01	9.368605e-01	7.678820e-02	BETTER
python-bug-69783-69784	Repair time	5.933612e-02	9.736065e-01	3.327286e-02	BETTER
	NCP	4.685328e-02	9.792545e-01	2.639350e-02	BETTER
wireshark-bug-37112-37111	Repair time	2.841686e-02	9.875338e-01	1.615646e-02	BETTER
	NCP	1.245814e-02	9.946122e-01	7.184645e-03	BETTER

equipped with FLFC are apparently smaller than the GenProg equipped with the single Jaccard module. It shows that FLFC improves the repair performance in both effectiveness and efficiency. More specifically, the average decrease of the repair time is 39.74% against using the single Jaccard module. It demonstrates that FLFC can significantly speed up the automatic repair process. Furthermore, the average decrease of NCP is 38.37%, and it shows that FLFC can considerably reduce the number of invalid patches generated.

Although the boxpolts provides a visual comparison, the result is not rigorous enough. Therefore, we further conduct a more rigorous and scientific comparison: the paired Wilcoxon-Signed-Rank Test [16]. The study performs two paired Wilcoxon-Signed-Rank Test: the repair time of 20 times on each version using FLFC v.s. that of using single Jaccard, and the NCP of 20 times on each version using FLFC v.s. that of using single Jaccard. Each test uses both the 2-tailed and 1-tailed checking at the σ level of 0.05.

Table 2 illustrates the p value of Wilcoxon-Signed-Rank test on repair time and NCP by comparing FLFC with Jaccard. As shown in Table 2, FLFC obtains BETTER results over Jaccard on all versions. It means that the repair time and NCP using FLFC significantly tends to be less than using Jaccard.

Based on the above results of the boxplots and Wilcoxon-Signed-Rank test, we conclude that FLFC is effective to improve the automatic repair performance.

4. Conclusion

From the perspective of automatic repair, this paper proposes a fault localization approach using failure-related contexts to improve the repair performance. We apply our approach to the state-of-the-art automatic repair technique GenProg, and compare our approach with the best fault localization technique in GenProg. The experimental results show that constructing failure-related contexts is effective to guide the follow-up patch generation activities and finally improve the automatic repair performance.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under Grant (Nos. 61379054 and 91118007).

References

[1] J. Anvik, L. Hiew, and G.C. Murphy, “Who should fix this bug?,” International Conference on Software Engineering, pp.361–370, 2006.

[2] R.C. Seacord, D. Plakosh, and G.A. Lewis, Modernizing legacy systems: Software technologies, engineering processes, and business practices, Addison-Wesley Professional, 2003.

[3] B. Hailpern and P. Santhanam, “Software debugging, testing, and verification,” IBM Systems Journal, vol.41, pp.4–12, 2002.

[4] A. Arcuri, “On the automation of fixing software bugs,” International Conference on Software Engineering, pp.1003–1006, 2008.

[5] A. Arcuri, “Evolutionary repair of faulty software,” Applied Soft Computing, vol.11, pp.3494–3514, 2011.

[6] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” 35th International Conference on Software Engineering, pp.802–811, 2013.

[7] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, “A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each,” 34th International Conference on Software Engineering (ICSE), pp.3–13, 2012.

[8] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” IEEE Trans. Softw. Eng., vol.38, no.1, pp.54–72, 2012.

[9] Y. Qi, X. Mao, Y. Lei, and C. Wang, “Using automated program repair for evaluating the effectiveness of fault localization techniques,” Proc. 2013 International Symposium on Software Testing and Analysis, ISSTA 2013, pp.191–201, New York, NY, USA, 2013.

[10] Z. Zhang, X. Mao, Y. Lei, and P. Zhang, “Enriching contextual information for fault localization,” IEICE Trans. Inf. & Syst., vol.E97-D, no.6, pp.1652–1655, June 2014.

[11] L. Naish, H.J. Lee, and K. Ramamohanarao, “A model for spectra-based software diagnosis,” ACM Trans. Software Engineering and Methodology (TOSEM), vol.20, no.3, p.11, 2011.

[12] M. Weiser, “Program slicing,” IEEE Trans. Softw. Eng., vol.10, pp.352–357, 1984.

[13] B. Korel and J. Laski, “Dynamic program slicing,” Information Processing Letters, vol.29, no.3, pp.155–163, 1988.

[14] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue, “Experimental evaluation of program slicing for fault localization,” Empirical Software Engineering, vol.7, no.1, pp.49–76, 2002.

[15] X. Zhang, N. Gupta, and R. Gupta, “A study of effectiveness of dynamic slicing in locating real faults,” Empirical Software Engineering, vol.12, no.2, pp.143–160, 2007.

[16] G.W. Corder and D.I. Foreman, Nonparametric statistics for non-statisticians: A step-by-step approach, John Wiley & Sons, 2009.