2018 IEEE International Conference on Software Maintenance and Evolution

An Empirical Study on the Effect of Dynamic Slicing on Automated Program Repair Efficiency

Anbang Guo, Xiaoguang Mao*, Deheng Yang, and Shangwen Wang
National University of Defense Technology
Changsha, China
{guoanbang12, xgmao}@nudt.edu.cn, {deheng_yang, shang_wen_wang}@163.com

***Abstract*—Research on the characteristics of error propagation can guide fault localization more efficiently. Spectrum-based fault localization (SFL) and slice-based fault localization are effective fault localization techniques. The former produces a list of statements in descending order of suspicious values, and the latter generates statements that affect failure statements. We propose a new dynamic slicing and spectrum-based fault localization (DSFL) method, which combines the list of suspicious statements generated by SFL with dynamic slicing, and take the characteristics of error propagation into account. To the best of our knowledge, DSFL has not yet been implemented in automated repair tools. In this study, we use the dynamic slicing tool Javaslicer to determine the error propagation chain of faulty programs and the statements related to failure execution. We implement the DSFL algorithm in the automated repair tool Nopol and conduct repair experiments on dataset Defects4j to compare the effects of SFL and DSFL on the efficiency of automated repair. Preliminary results indicate that the scope of error propagation for most programs is a single class, and the DSFL makes automated repair more efficient.**

Keywords*—Automated program repair, dynamic slicing, spectrum-based fault localization, error propagation

I. INTRODUCTION

Automated program repair is the process of automatically repairing programs. Research on automated program repair has attracted considerable attention in the field of software maintenance, and many tools have been proposed [4], [5], [6], [7], [8].Test-suite based repair is currently the main method used for automated program repair, which includes three phases: fault localization, patch generation and patch validation. Fault localization is the first step in the process of automated program repair. Spectrum-based fault localization (SFL) techniques are the main fault localization techniques used by automated program repair tools [9], [10]. Slice-based fault localization techniques have also been widely investigated in the field of fault localization [11].The accuracy of fault localization will directly affect the efficiency of patch generation.

*Corresponding Author

In SFL techniques, the suspicious values of the statements are calculated by comparing the coverage information of program elements, such as statements, branches, and basic blocks in failed and successful executions. The main idea is that if a program element is covered with many failed executions, but rarely covered with successful executions, then this program element may contain faults [10]. The advantage is the provision of suspicious values of the statement and the low complexity. Slice-based fault localization techniques have also shown several advantages: such as describing the context that causes the failure and requiring only the failure test case. Program slicing includes static slicing and dynamic slicing. Studies have shown that dynamic slicing works better than static slicing for fault localization [11]. However, program slicing cannot provide a description of the suspiciousness of the statement. Recently, researchers have proposed combining the advantages of program slicing and program spectrum to improve the efficiency of fault localization [12].

Although the dynamic slicing and spectrum-based fault localization(DSFL) technology has been proposed recently[12], to the best of our knowledge, no automated program repair tool has used this fault localization method. In addition, unlike the previously proposed DSFL, we also take the characteristics of error propagation into account to form a new DSFL method. Thus, an empirical study on automated program repair should be conducted using this technology to explore its effectiveness.

In recent work, we manually analyzed the patches generated by programmers and bug reports of 197 program versions in three real-world large projects, i.e. JfreeChart, Commons Lang and Commons Math. We call the patches generated by the programmers as manual patches. We found that in 84.3% of the program versions: the modified statement of the manual patch and the failure statement reported in the bug reports are in the same classes. We also determined that the error is spread in a single class by analyzing the error propagation chain (EPC). Therefore, we consider the statements in both the failure class reported in the bug report and the program slicing as the final program slicing result. In this manner, the range of suspicious statements is further reduced.

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00066

In the present study, we compare the efficiency of SFL and DSFL using two metrics: the number of candidate patches (NCP) generated before a valid patch is identified [14] and the repair time. We select Javaslicer, a dynamic slicing tool [15], and Nopol, a tool for Java program repair [8], as our experimental tools, and perform experiments on eight buggy programs from Defects4J, which is a bug dataset extensively used for evaluating Java program repair systems [16].

In summary, this study presents the following contributions:

- We manually analyze the manual patches and bug reports of 197 program versions, and use Javaslicer to obtain the EPC of 10 versions.
- We conduct an empirical study on the usage of program dynamic slicing in automated program repair, and a new DSFL method is implemented in Nopol, an SFL-based repair tool, to compare the efficiency of the DSFL and SFL methods. We also use the two repair tools to perform repair on the Defects4j programs.
- Experimental results suggest that (1) The error propagation of most faulty programs is in a single class, and (2) the DSFL makes automated program repair more efficient than that of the SFL.

The remainder of this paper is organized as follows. Section II describes related works on error propagation and fault localization techniques. Section III presents our experimental design in detail. Section IV discusses our results. Section V provides the conclusion and future work.

II. Related work

Fault localization aims to determine the location of defect in the program, and repair aims to fix the exposed program error to avoid failure. Defects are activated under certain conditions, thereby causing errors in the program operation. The error propagation eventually leads to failure. Therefore, the characteristics of error propagation before conducting fault localization should be examined [17]. The error propagation chain (EPC) is a sequence of statements between program defect and program failure statement.

SFL techniques are currently the most widely used techniques. In SFL, the program is represented as a program spectrum. Each bit of program spectrum corresponds to a statement in the program. The suspiciousness of each statement can be calculated according to the collected program operation data. The program statements are sorted by their suspiciousness. Dynamic slicing analyzes the program execution path for a given input and identifies the set of statements that can affect the given program statements. The DSFL considers the program slicing information and the characteristics of error propagation based on the SFL to improve the efficiency of fault localization.

For example, in a faulty program P, we assume that, in execution of test case t_2, the bug report shows that $Statement_{15}$ is the failure statement, but the failure is actually caused by $Statement_9$. In addition, $Statement_{13}$ is not in the failure class reported in the bug report. A sequence of statements generated by the SFL techniques for a faulty program is as follows: {($Statement_{18}$, 0.9), ($Statement_1$, 0.8), ($Statement_{13}$, 0.8), ($Statement_9$, 0.7), ($Statement_{15}$, 0.6), …}. ($Statement_9$, 0.7) indicates that the suspicious value of $Statement_9$ is 0.7. The statement set obtained by dynamic slicing on $Statement_{15}$ is {$Statement_{15}$, $Statement_{13}$, $Statement_{10}$, $Statement_9$, $Statement_2$}. After removing $Statement_{13}$, the final slicing result is {$Statement_{15}$, $Statement_{10}$, $Statement_9$, $Statement_2$}. Thus, the statement sequence generated by the DSFL is {($Statement_9$, 0.7), ($Statement_{15}$, 0.6)}. Evidently, the DSFL identifies the faulty statement $Statement_9$ earlier than that of the SFL.

To study the characteristics of error propagation, we manually analyze 197 program versions of manual patches and bug reports. Afterward, we select 10 of these program versions and obtain their EPCs. We implement the DSFL in Nopol which originally uses the SFL, and design experiments to compare the efficiency of DSFL-based Nopol and SFL-based Nopol. The experimental results indicate the characteristics of error propagation and the differences between DSFL and SFL, and provide a reference for future research on fault localization techniques in automated program repair.

III. Experiment Design

A. Tools and dataset

We select Javaslicer [1], [15], a dynamic slicing tool, to obtain the EPC and statements related to failure. The two reasons for selecting Javaslicer are as follows: firstly, dynamic slicing is better than static slicing for fault localization. Secondly, unlike other dynamic slicing tools, Javaslicer is open source and widely used [18], [19].We analyze the manual patches and bug reports of faulty programs, and assume that the modified statement (r) in the manual patch is the root cause of the failure, and the statement reported in the bug report is the failure statement (o). The EPC is obtained by using Javaslicer, as described in Algorithm 1. Firstly, the test case that causes the failure is run, and the execution statements are collected. Subsequently, the dynamic slicing DS (r) and DS (o) of r and o are obtained respectively. The EPC is the intersection of DS (r) and DS (o).

Algorithm 1: EPC Algorithm

1: **Input:** root cause r, failure statement o, program P, failed testcase t_i
2: **Output:** EPC(r, stm_1, stm_2, … , stm_n, o)
3: ES = GetFailedExecutedStatements(P, t_i)
4: DS(r) = GetDynamicSlice(r)
5: DS(o) = GetDynamicSlice(o)
6: EPC = DS(r) ∩ DS(o)

After analyzing the characteristics of error propagation, we implement DSFL in Nopol [2], as shown in Algorithm 2, to compare the DSFL with the SFL. Given a set of test cases and a faulty program P, Nopol uses the SFL to obtain a list of suspicious statements. Failure classes and failure statements are obtained from the bug report generated by executing the failed test case t_i. Javaslicer is utilized to identify the program slices affecting failure statements. According to the characteristics of error propagation, we only select the state-

Algorithm 2: DSFL Algorithm

```
Input: Testcases {t1,t2…tn}, Faulty program P
Output:
DSFLList {(stm1,susp1),(stm2,susp2)…(stmn,suspn)}
RankList = GetSFLList (Testcases, P)
For all ti ∈Testcases do
   if TestResult(ti) = failure then
      FC[i] = GetFailureClass (report)
      FS[i] = GetFailureStm (report)
      AllSlice = AllSlice ∪GetDSlice (ti, FC[i], FS[i])
   end if
end for
      Slice = AllSlice∩StmOfFC
For all stmi∈RankList do
   if stmi∈Slice then
      DSFLList = DSFLList∪{(stmi,suspi)}
   end if
end for
```

ments that are both in the program slicing and failure class as the final program slicing result. Afterwards, we combine the RankList obtained by the SFL with the program slicing results and finally generate the list of suspicious statements produced by the DSFL. Patch generation and patch verification are also performed on the basis of the list of suspicious statements generated by the DSFL.

Additionally, we select Defects4J [3]，a large, peer-reviewed dataset of real-world Java bugs, as our experimental benchmark. Defects4J contains 357 real bugs from five real-world open source programs to enable reproducible studies in software testing research [16]. Recently, Defects4j is widely used in automated repair to compare the effectiveness of various repair tools and fault localization techniques [20].To date, we have performed experiment on eight faulty programs: Chart_5, Chart_9, Chart_13, Chart_17, Math_40, Lang_44, Lang_51, and Lang_58. We will conduct experiments on more programs from Defects4j in the future.

B. Evaluation Metrics

In this study, we use two evaluation metrics to assess the effect of program dynamic slicing on repair efficiency.

- *NCP:* The number of candidate patches generated before a valid patch is identified. This metric has been used to evaluate the repair effectiveness of automated repair tools [3], [14].The smaller the value of NCP is, the better effectiveness of fault localization algorithm is. The lists of suspicious statements generated by the two algorithms are different, which results in different NCP during the repair phase.

- *Repair Time:* Repair time starts from the localization process until a valid patch is identified or until time is out. The repair time consists of localization time, candidate patch generation time, and patch verification time. The time spent by the SFL and DSFL in generating the list of suspicious statements is different and different lists of suspicious statements are produced by the two algorithms. Therefore, these algorithms will lead to different repair times. Moreover, previous studies have used this metric to evaluate the effectiveness of automated program repair tools [21].

C. Implementation of the Experiment

In the SFL techniques, many suspiciousness calculation formulas have been proposed. To study the effect of dynamic slicing on program repair under different SFL techniques, we implement two formulas, including Tarantula and Jaccard [13], in Nopol. Given the faulty program, the SFL techniques collect the execution trace of the positive and negative test cases, and use the calculation formula to compute the suspiciousness of each statement. The two formulas are shown in Table I.

We also apply the DSFL to the previously implemented suspiciousness-first algorithm (SFA)-based Nopol and rank-first algorithm (RFA)-based Nopol [22], which originally use the SFL, to assess the influence of SFL and DSFL on program repair under different statement selection strategies. The RFA selects the top-ranked statement as the target statement. In the SFA, the probability that each statement will be selected for modification depends on its suspiciousness value.

TABLE I. CALCULATION FORMULAS

Formula Name	Algebraic Form
Tarantula	$\frac{a_{ef}}{a_{ef}+a_{nf}}$
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$

a_{ef} is the number of failed tests executing the statement.
a_{nf} is the number of failed tests that do not execute the statement.
a_{ep} is the number of passed tests executing the statement.
a_{np} is the number of passed tests that do not execute the statement.

We implement these two suspiciousness calculation formulas in the Nopol version based on the RFA and SFA. The DSFL algorithm is also implemented in the four versions of the Nopol. In this way, we finally obtain 8 Nopol versions. To ensure the accuracy of the experimental results, we use these 8 versions of Nopol to perform 100 repeated repairs for each faulty program. The evaluation metrics for the repair results are presented in boxplots, and analyzed by the Wilcoxon signed-rank test. Finally, the effect of dynamic slicing on the efficiency of automated repair can be summarized according to these data.

IV. RESULTS AND ANALYSIS

A. Experimental results

We analyzed 197 versions of 3 programs in Defects4j. As shown in Table II, in 166 versions, the root cause of program failure and the failure statement reported in the bug report are in the same class. With the use of a sampling method, we extract 10 of the 166 programs and implemented algorithm 1 to obtain their EPC.

TABLE II. RATE OF ERROR PROPAGATION IN SINGLE CLASS

Program	Rate
Chart	25/26
Lang	62/65
Math	79/106
Total	166/197

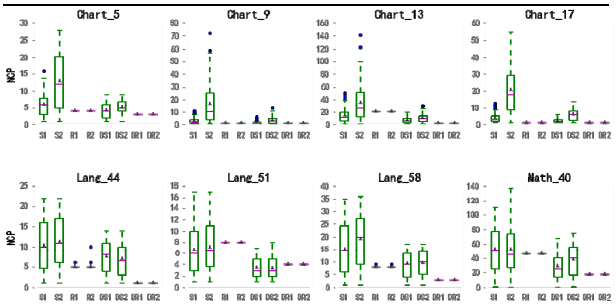

Fig. 1. Boxplot of NCP. S1 and S2 represent the SFL-based Nopol, which uses SFA, with the implementation of the Jaccard and Tarantula formulas, R1 and R2 represent the SFL-based Nopol, which use RFA, with the implementation of Jaccard and Tarantula formulas. DS1, DS2, DR1 and DR2 represent the DSFL-based Nopol implementing the Jaccard and Tarantula formulas, which use SFA and RFA, respectively.

TABLE III. MEAN NCP OF 8 NOPOL VERSIONS

	P-Value S1&DS1	P-Value S2&DS2	P-Value R1&DR1	P-Value R2&DR2
Chart_5	0.0004	0.0000	0.0000	0.0000
Chart_9	0.0012	0.0000	**1.0000**	**1.0000**
Chart_13	0.0000	0.0000	0.0000	0.0000
Chart_17	0.0000	0.0000	**1.0000**	**1.0000**
Lang_44	0.0047	0.0000	0.0000	0.0000
Lang_51	0.0000	0.0000	0.0000	0.0000
Lang_58	0.0001	0.0000	0.0000	0.0000
Math_40	0.0000	0.0074	0.0000	0.0000

At Significance level of 5%.

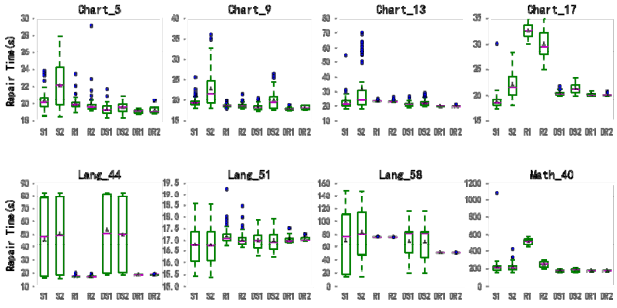

Fig. 2. Boxplot of Repair Time.

Figure 1 shows the distribution of NCP obtained by 100 repeated repairs for each faulty program on 8 Nopol versions.

TABLE IV. MEAN REPAIR TIME OF 8 NOPOL VERSIONS

	P-Value S1&DS1	P-Value S2&DS2	P-Value R1&DR1	P-Value R2&DR2
Chart_5	0.0000	0.0000	0.0000	0.0000
Chart_9	0.0000	0.0000	0.0000	0.0000
Chart_13	**0.1136**	0.0006	0.0000	0.0000
Chart_17	0.0000	**0.1361**	0.0000	0.0000
Lang_44	0.0008	**0.0536**	0.0000	0.0000
Lang_51	0.0143	0.0152	0.0019	0.0005
Lang_58	**0.0533**	**0.5252**	0.0000	0.0000
Math_40	0.0000	0.0000	0.0000	0.0000

At Significance level of 5%.

The boxplot shows the differences between SFL-based Nopol and DSFL-based Nopol on NCP. To further evaluate the effectiveness of the two algorithms, we perform the Wilcoxon signed-rank test to calculate the p-values of NCP generated by the SFL and DSFL algorithms, as shown in Table III.

Figure 2 depicts the repair time of each faulty program for eight Nopol versions. The distribution of repair time in Figure 2 is similar to that in the boxplots of NCP in Figure 1. We also calculate the p-values of repair time to further analyze the effectiveness of the SFL and DSFL algorithm, as displayed in Table IV.

B. *Analysis*

a) The error propagation of most faulty programs is in a single class: As shown in Table II, the failure and root statement that cause the failure are in the same class, and they account for 84% of the 197 faulty programs. When the EPCs of the 10 extracted sample programs are obtained, the errors of all 10 progams are only propagated in single class. This class is the failure class reported in the bug report. Thus, we can conclude that the majority of faulty programs spread the error in a single class, such that we can consider only the failure class reported in the bug report when performing debugging or fault localization.

b) The DSFL makes automated program repair more efficient than that of the SFL: As shown in Figure 1, the mean and minimum NCP of DSFL-based Nopol are smaller than or equal to the mean and minimum NCP of SFL-based Nopol for the same faulty program, regardless of any suspiciousness calculation formulas and statement selection rule. This finding indicates that the DSFL-based Nopol modifies fewer suspicious statements before generating a valid patch. Table III further confirms that the repair efficiency of DSFL-based Nopol is significantly improved under the NCP metric. The DSFL considers the dynamic slicing information, the list of suspicious statements produced by the SFL, and the error propagation characteristics, thereby resulting in a new list of suspicious statements. Consequently, the search space for statement selection during candidate patch generation is reduced.

On the basis of the results of repair time, we determine that in most cases, the time required by the DSFL-based Nopol to repair the same faulty program is less than that of the SFL-based Nopol. On the contrary, in a few cases, the time required by SFL-based Nopol to repair faulty program is less. The DSFL reduces the search space for the candidate patch generation and patch verification phases, such that fewer statements are considered and less time is spent before producing a valid patch. However, the DSFL also needs to collect dynamic slicing information for the faulty programs. Furthermore, longer time may be needed to collect dynamic slicing information than that for patch generation and patch verification on the statements removed by DSFL from the list generated by SFL. Thus, in a few cases, the DSFL-based Nopol may take more repair time than that of the SFL-based Nopol. Wilcoxon test results show that significant difference exists between SFL and DSFL in terms of repair time of 8 Nopol versions at a significance level of 5%. In summary, the DSFL-based Nopol is more efficient at repairing the same faulty program than that of the SFL-based Nopol when the metric is repair time.

Analysis on NCP and repair time demonstrates that DSFL exhibits a significant improvement in the efficiency of automated program repair compared with that of SFL.

V. Conclusion and Future Work

In this paper, we analyzed the manual patches and bug reports of 197 faulty programs and use the dynamic slicing tool Javaslicer to obtain the EPC of 10 programs. On the basis of the EPC and analysis results, we can conclude that the error propagation of most faulty programs is in a single class. Accordingly, we should only consider the statements in the failure class reported in the bug report when performing fault localization.

Moreover, we implement a new DSFL algorithm in RFA-based Nopol and SFA-based Nopol [22] with two suspiciousness calculation formulas, which originally use SFL. Afterwards, we use these 8 versions of Nopol to perform 100 repeated repair experiments on 8 faulty programs from Defects4j. The effects of DSFL and SFL on automated repair efficiency are evaluated under our evaluation metrics, namely, NCP and repair time. The preliminary experimental results show that DSFL-based Nopol is more efficient than SFL-based Nopol in repairing the same faulty program under the metric of NCP and repair time. Therefore, we can conclude that dynamic slicing makes fault localization and automated program repair more efficient.

In future work, we plan to obtain the EPC for a large number of faulty programs and provide basis to further improve the accuracy of fault localization and the correctness of automated repair by analyzing the characteristics of error propagation. We also intend to conduct more repair experiments on more faulty programs in Defects4j with more automated program repair tools to confirm our findings.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (Grant No.61379054, 61672592).

References

[1] https://github.com/hammacher/javaslicer

[2] https://github.com/SpoonLabs/nopol

[3] https://github.com/rjust/defects4j

[4] Le Goues, Claire, et al. " Genprog: A generic method for automatic program repair. " *IEEE Transactions on Software Engineering* 38.1: 54-72,2012.

[5] Kim, Dongsun,et al. " Automated patch generation learned from human-written patches. " *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

[6] Qi Yuhua, et al. " The strength of random search on automated program repair. " *Proceedings of the International Conference on Software Engineering*. IEEE Press, 2014.

[7] Tan, Shin Hwei, and Abhik Roychoudhury. " reflix: Automated repair of software regressions. " *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015.

[8] Xuan Jifeng, et al. " Nopol: automatic repair of conditional statement bugs in java programs. " *IEEE Transactions on Software Engineering* 43.1:34-55, 2017.

[9] Naish L, Lee, Ramamohanarao K. " A model for spectra-based software diagnosis. " *ACM Transactions on Software Engineering and Methodology* 20.3:1-32,2011.

[10] Xie X, et al. " A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. " *ACM Transactions on Software Engineering and Methodology* 22.4:402-418,2013.

[11] Zhang X, Gupta N, Gupta R. " Pruning dynamic slices with confidence. " *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2006.

[12] Mao Xiaoguang, et al. " Slice-based Statistical Fault Localization. " Journal of Systems and Software 89:51-62, 2014.

[13] Wong, W. Eric, et al. " A survey on software fault localization. " *IEEE Transactions on Software Engineering* 42.8:707-740,2016.

[14] Qi, Yuhua, et al. "Using automated program repair for evaluating the effectiveness of fault localization techniques." *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013.

[15] Hammacher C, Streit K, Hack S and Zeller A. " Profiling java programs for parallelism. " The Workshop on Multicore Software Engineering. 2009.

[16] Just, René, Darioush Jalali, and Michael D. Ernst. " Defects4J: A database of existing faults to enable controlled testing studies for Java programs. " *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 2014.

[17] Zeller A. " Why Programs Fail: A Guide to Systematic Debugging " 2nd Edition. Elsevier, 2009.

[18] Xuan J and Monperrus M. " Test case purification for improving fault localization. " *Proceedings of the 22nd ACM SIGSOFT International Symposium on foundations of software Engineering*. ACM, 2014.

[19] Zhang Y and Mesbah A. " Assertions are strongly correlated with test suite effectiveness. " *Proceedings of the 2015 joint meeting on Foundations of Software Engineering*. 2015.

[20] Durieux, Thomas, et al. " Automatic repair of real bugs: An experience report on the defects4j dataset. " 2015.

[21] Nguyen, Hoang Duong Thien, et al. "Semfix: Program repair via semantic analysis." *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013.

[22] Yang Deheng, Qi Yuhua, Mao Xiaoguang. " An Empirical Study on the Usage of Fault Localization in Automated Program Repair. " *Proceedings of 2017 IEEE International Conference on Software Maintenance and Evolution*. IEEE Press,2017.