Fix Director for Automated Program Repair

Juhyoung Kim
Department of Computer Science and Engineering
SungKyunkwan University
Suwon, Republic of Korea
kjhkjh612@skku.edu

Eunseok Lee
Department of Computer Science and Engineering
SungKyunkwan University
Suwon, Republic of Korea
leees@skku.edu

***Abstract*—As the scale of software increases, automated program repair (APR) is required as an essential technology to reduce bug-fixing manual effort. One of the most effective APR approaches is the template-based APR. Although this data-driven approach has made many improvements in fixing bugs, there is still a lack of information utilization at each stage of the program repair. In this paper, we propose a fix pattern prioritization APR technique. We do not just use the test cases to determine whether the source file is fault or not, but as a material for why the bug occurred. With this information, we select a more appropriate fix action and apply it first to increase the efficiency of the automated program repair.**

***Keywords*—Software engineering, Automated program repair, Fix pattern prioritization**

I. Introduction

The scale of modern software developments are growing. As a result, the tasks of maintaining and repairing software have been consuming more time and cost. Recent study showed that these process account for about 50% of the software development [1]. To reduce this excessive cost, the demand for system to automatically fix software defects has increased. In this aspect, various approaches of automated program repair techniques have been proposed. Most of them are search-based APR [2, 3, 4, 5, 6, 7, 8]. They generate patch candidates using predefined sets of mutation operators with fault space determined by Fault Localization (FL) techniques. And they search correct patch that passes all given test cases among generated patch candidates from APR techniques. Despite this novel approach, there are some problems. First, search space to generate correct patch is usually huge. Second problem is correct patches may not exist in the search space, so APR techniques cannot fix the program. In addition, even if we increase the search space to find the correct patch, we are not sure that we can find the correct patch. Rather, it can only cause an increases cost in time and effort. To solve this problem, other approaches have been proposed including template-based [9,10,11,12,13,14], machine learning-based [15,16,17,18], etc. In particular, template-based APR techniques have been evaluated as the most effective and superior performance techniques [19]. This approach is strategy of APR to generate concrete patches based on fix patterns (also referred to as fix templates) which are predefined using human written patches from open source repositories. Because these approaches leverage predefined template from human-written patches, we can find more correct patches and reduce the cost. Despite these efforts, APR systems still do not have enough effectiveness for the cost it puts in. For example, many techniques including template-based APR go through three steps to repair the program. First step is the fault localization (FL), which identifies the fault Location. The second step is to generate candidate patches based on the proposed approaches in various ways. Final step is the validation, which checks the candidate patch that passes all given test cases. In this process, information used at individual step is often used only at each step and discarded. Specifically, the occurrence of failed test cases is used to check whether the source code has defects, but the information of the failed test cases is rarely used when going through other repair process. This disconnection in utilizing information can be a critical loss for program repair. In this paper, we leverage the information of the failed test cases to overcome disconnection in utilizing collected information. Using information of the failed test cases, we can infer what is the cause making defects and find a basis for which fix pattern should be prioritized in proceeding program repair. And these processes achieve performance improvements that reduce the cost.

II. Background and Motivation

A. Template-based APR

Template-based APR techniques generate bug-fixing candidate patches automatically using fix patterns extracted from human written patches. Existing template-based APR

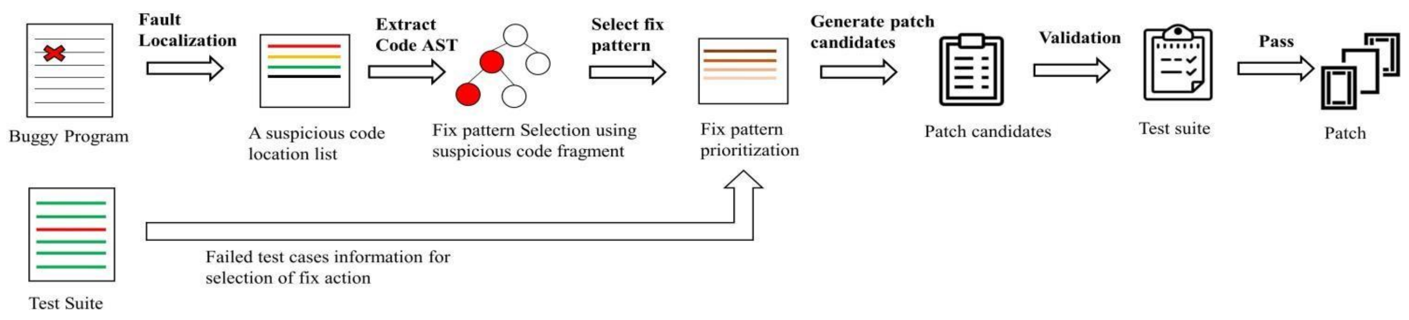
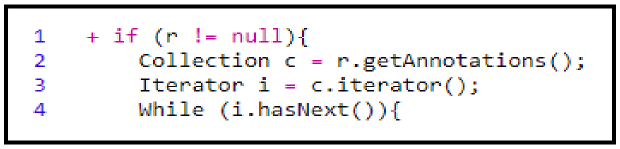

Fault Localization
Extract Code AST
Select fix pattern
Generate patch candidates
Validation
Pass
Buggy Program
A suspicious code location list
Fix pattern Selection using suspicious code fragment
Fix pattern prioritization
Patch candidates
Test suite
Patch
Test Suite
Failed test cases information for selection of fix action

Fig. 1. Workflow of Fix director

techniques have own method to extract fix pattern and fix pattern classification. The progress of the template-based APR techniques is as follows. Template-based APR first identifies fault locations (e.g., statements and expressions) and uses the fix patterns that researcher extracted before to generate patch candidates. Finally, validate the patch candidate checking generated patch passes all given test cases. This process reduces the cost and increase the possibility of finding the correct patch over traditional approaches, (i.e., search-based APR).

B. *Fix pattern selection*

Existing template-based techniques have own approaches to select the fix patterns. CAPGEN [14] uses the frequency information of fix pattern and similarity information for suspicious fault location. TBar [19] uses the child node information of the suspicious fault location to select fix pattern. Many template-based APR techniques including above techniques use their own approaches using fault statement information to select fix patterns. But there is little consideration as to why this statement is wrong. They mainly utilize information suspicious statement e.g., (structure, node type, variable name). On the other hand, in case of developers, if the fault location is identified in the code, they diagnose why the source code is wrong using given failure history. Using failure information, developers repair the program. If APR techniques are able to obtain and utilize information from the failed test cases, it will be easier to determine what fix action to take. Fig 2 shows a defect, Chart-4, in Defects4j [22] benchmark. Line 2 is given as the bug line, then APR tools fix this code using given code snippet`s information. Most template-based APR techniques modify the program using selected fix patterns without specifying which problems to fix. This leads to a significant waste of resources and time. However, if we use information about the NullPointerException error message generated from test suite, we can obtain the direction of fix action. This makes program repair process easier.



```
1   + if (r != null){
2       Collection c = r.getAnnotations();
3       Iterator i = c.iterator();
4       While (i.hasNext()){
```

Fig. 2. The faulty code from Chart-4

III. APPROACH

Figure 1 shows an overview workflow of our approach. Fault localization is the first step. It identifies suspicious code locations where we apply the fix action. In our experiment, we used the perfect localization which means fault statement line number is given to reduce the FL noise. When the fault localization step makes a list of suspicious location, we try to select the fix pattern extracted from human patch written. In this process we utilize two different pieces of information, first is the suspicious statement context information. We traverse each child node of the suspicious statement AST and try to match each node with fix pattern predefined. Multiple fix patterns can be generated for each suspicious statement. Since classifying fix patterns works are similar with classifying the behavior of developers who fix bugs, considerations for bug type needs to be given to determine which fix action is more appropriate. This work is implemented by preferentially applying a more appropriate fix pattern for the bug, utilizing the information from the collected failed test case error messages. In case of null pointer exception, fix patterns that are more relevant to null pointer should be applied first. In addition, we prioritize fix patterns that should be taken for eight error types, such as when the wrong value was calculated or when the wrong index was referenced. Then, we applied this approach to 52 bugs generated correct patches using TBar which is the most effective template-based APR tool. And we measure changes in the rank of fix pattern that generated the correct patch. Table 1 shows the rank without prioritization and with prioritization results. Each pattern generates at least one to dozens of candidate patches. These pattern prioritization tasks are designed to generate the correct candidate patches faster and follow the process of real developers fixing bugs. Because selecting fix patterns is the same process as choosing what kind of fix action developers take, utilizing error messages as a basis for selecting fix pattern can be reasonable decision making.

TABLE I. FIX PATTERN RANK

BugId	Pattern rank that first correct patch appear	
	Without prioritization	***With prioritization***
Chart_1	1	1
Chart_4	2	1
Chart_8	1	1
Chart_9	2	1
Chart_11	1	1
Chart_12	5	5
Chart_19	1	1
Chart_20	1	1
Chart_24	2	1
Chart_26	2	2
Closure_2	2	1
Closure_4	1	1
Closure_10	1	1
Closure_11	1	1
Closure_13	3	3
Closure_38	1	1
Closure_40	3	3
Closure_46	1	1
Closure_62	1	1
Closure_70	4	3
Closure_73	1	1
Closure_102	4	4
Closure_115	5	5
Closure_117	3	5
Lang_6	2	2
Lang_10	4	4
Lang_24	1	1
Lang_26	3	2
Lang_33	2	1
Lang_39	2	1
Lang_47	3	1
Lang_51	3	3
Lang_57	2	2
Lang_59	1	1
Math_4	2	1
Math_5	1	1
Math_11	2	2
Math_57	1	1
Math_58	1	1
Math_65	4	3
Math_70	1	1
Math_75	1	1
Math_77	1	1
Math_79	1	2
Math_82	1	1
Math_85	1	1
Math_89	4	1
Mockito_26	3	1
Mockito_29	1	1
Mockito_38	2	1
Chart_1	1	1
Chart_4	3	1

🟦 Indicates that prioritization improves the pattern ranking

🟥 Indicates that prioritization degrades the pattern ranking

IV. Evaluation

A. *Experiment Setup*

For evaluating our approach, we select the Defects4J dataset as the evaluation benchmark. This benchmark is a widely used dataset and recent state-of-the-art APR systems targeting Java program defects in automatic program repair field. Defects4j consists of six large projects and contains 395 bugs in total. To reduce fault localization noise, we implement an experiment with information about buggy statement location.

We implement experiments in two ways. First experiment is a reimplementation of TBar which is the best effective APR technique in the world. TBar defines 35 code change patterns and apply different patterns according to the context of the bug statements. Therefore, several fix patterns are applied to one defect. When fixing bugs, we record the rank of the pattern that generates correct patch. The second experiment is using information from the failed test cases. And we prioritize fix patterns more related with bugs. And then we record the pattern rank as in the first experiment.

B. *Rsearch Questions and Results*

RQ1. How effective is the fix pattern prioritization?

We implement two experiments. First experiment is without fix pattern prioritization and the other is with fix pattern prioritization. Table 1 shows the results of two cases. The pattern ranking means pattern`s rank that first correct patch is found. For example, rank 1 means that correct patch is found in the first applied fix pattern. 6 means that it fails for the 5 fix patterns applied and succeed with the 6th fix pattern. In case of 15 bugs, the ranking of patterns has improved. Since multiple candidate patches are created in a single fix pattern, the change in fix pattern ranking 1 may be more change in patch candidate ranking.

RQ2. Can failed test cases information be fix ingredients for APR?

We collect 8 failed message types. For each type of errors, we match appropriate fix patterns for error message types and apply them to prioritization process. This process is similar with developer`s bug fixing process when they manually fix bugs. Because developer checks the error information and choose their fix behavior using collected bug information. Using this approach, we can get performance improvements as seen in Table1. As a result, Information on what type of errors can be the basis for which fix action should be chosen.

RQ3. Failed test cases information covers all cases?

Error messages do not cover bugs in all cases. In actual codes, there are quite a few cases where only the Assertion message appears. Additionally, there are no information about the error appears. In these cases, we are not able to prioritize which fix action to take because we do not have the additional error information.

V. CONCLUSION

Template-based APR techniques based on fix patterns have been studied in various approaches to fix the bug program. Although template-based APR techniques have been known as the most effective approach, there is still a lack of information utilization between the entire processes. APR is still less efficient than developers fixing bugs manually. Because most APR technologies focus on bug location to fix the bugs without consideration as to why the bugs occur, we may miss important information to repair the program. To overcome this, we try to prioritize fix action based on why the bug occurred in this study. Using information for bugs, we can get correct patch earlier. In future APR studies, it is necessary to increase the efficiency by using information on the location of bug as well as information on why bugs occur.

VI. FUTURE WORK

We conduct fix pattern prioritization for eight error message types. However, there are more error messages that need to be classified. Because there are more various error messages in real code. In some cases, it is difficult to judge only by the information in the error message in determining what defects are present in the actual code. Therefore, we need additional information on what causes the bugs. Thus, future research will be conducted with the aim of complementing the above limitations.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea grant funded by the Korea government (MSIT) (2019R1A2C2006411)

REFERENCES

[1] T. Britton, L. Jeng, G. Carver, and P. Cheak, ‘‘Reversible debug-ging software - quantify the time and cost saved using reversible debuggers,’’ 2013

[2] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering 38, 1 (2012), 54–72.

[3] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A largescale experiment on the defects4j dataset. Empirical Software Engineering, pages.

[4] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In Proceedings of the 31st International Conference on Software Engineering. IEEE, 364–374

[5] Vidroha Debroy and WEric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, 65--74.

[6] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering 20, 1 (2015), 176--205.

[7] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In ISSTA'2015. ACM, 24—36

[8] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 831– 841. ACM, 2017.

[9] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In Proceedings of the 35th International Conference on Software Engineering. IEEE, 802–811.

[10] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. FixMiner: Mining relevant fix patterns for automated program repair Empirical Software Engineering volume 25, pages1980–2024(2020)

[11] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation. IEEE.

[12] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering.IEEE.

[13] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In Proceedings of the International Symposium on Search Based Software Engineering. Springer, 65–86.

[14] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair in ICSE.

[15] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In Proceedings of the 31st AAAI Conference on Artificial Intelligence. AAAI Press, 1345–1351.

[16] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering. IEEE.

[17] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, pages 832–837, 2018.

[18] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei and Lin Tan. CoCoNuT: Combining Context-Aware Neural Translation Models using Ensemble for Program Repair. In ISSTA 2020

[19] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. Tbar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, pages 31–42, New York, NY,USA,2019.ACM.

[20] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, 314–324.

[21] Rui Abreu, Arjan JC Van Gemund, and Peter Zoeteweij. 2007. On the accuracy of spectrum-based fault localization. In Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION. IEEE, 89–98.

[22] 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 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 437–440.

[23] Kui Liu, Koyuncu Anil, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for

Automated Program Repair. In Proceedings of the 25th Asia-Pacific Software Engineering Conference. 658–662

[24] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering.IEEE.

[25] Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. Empirical Software Engineering 14, 3 (2009), 286–315.

[26] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An empirical analysis of the influence of fault space on searchbased automated program repair. arXiv preprint arXiv:1707.05172 (2017).

[27] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In Proceedings of the 40th International Conference on Software Engineering. ACM, 789–799.

[28] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, Marcelo A. Maia. Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. SANER'18 (25th edition of IEEE International Conference on Software Analysis, Evolution and Reengineering)

[29] Mohammed A. Shehab, Yahya M. Tashtoush, Wegdan A. Hussien, Mohammed N. Alandoli, and Yaser Jararweh, "An Accumulated Cognitive Approach to Measure Software Complexity," Vol. 6, No. 1, pp. 27-33, February, 2015. doi:10.12720/jait.6.1.27-33