Fix Director for Automated Program Repair

Juhyoung Kim

Department of Computer Science and Engineering SungKyunkwan University

Suwon, Republic of Korea [kjhkjh612@skku.edu](mailto:kjhkjh612@skku.edu)

Eunseok Lee

Department of Computer Science and Engineering

SungKyunkwan University Suwon, Republic of Korea [leees@skku.edu](mailto: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 approach 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***

1. INTRODUCTION

The scale of modern software is getting larger. Accordingly, maintaining and repairing software have been time-consuming and expensive process. Recent study showed that these process account for about 50% of the software development [1]. To reduce this excessive consumption cost, the need for a system to automatically fix the software defects has become important.

In this aspect, various approaches of automated program repair techniques have been proposed. Many of them are search-based APR [2, 3, 4, 5, 6, 7, 8]. They generate patch candidate using predefined set of mutation operators with fault space determined by Fault Localization (FL) techniques. They search correct patch that passes all given test cases among these patch candidates.

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, only the cost of time and effort may increase.

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 of open sources. Because these approaches leverage predefined template from human-written patches, we can find more correct patch and reduce the cost.

However, despite these efforts, APR system still do not have enough effect for the cost it puts in. For example, many techniques including template-based APR go through three steps for program repair. First step is the fault localization (FL), which identifies the fault Location. The second step is generating the candidate patches based on respective proposed approaches. Final step is the validation, which checks the candidate patch that passes all given test cases. In this process, information used at individual steps is often used only in each step and discarded. Specifically, the occurrence of failed test cases is used to confirm that the source code has defects and the information of the failed test case is not used when other repair process. This disconnected use of information can be a critical loss to program repair.

In this paper, we leverage the information of the failed test cases. Using information of the failed test cases, we can infer what is the cause making defects and find a basis for which fix pattern could be prioritized in performing program repair and achieve performance improvements that reduce the cost.

1. BACKGROUND AND MOTIVATION
2. *Template-based APR*

Template-based APR generates bug-fixing candidate patches automatically using fix patterns extracted from human written patches. Existing template-based APR techniques have own method to extract fix pattern and fix pattern classification. The progress of the template-based APR is as follows. Template-

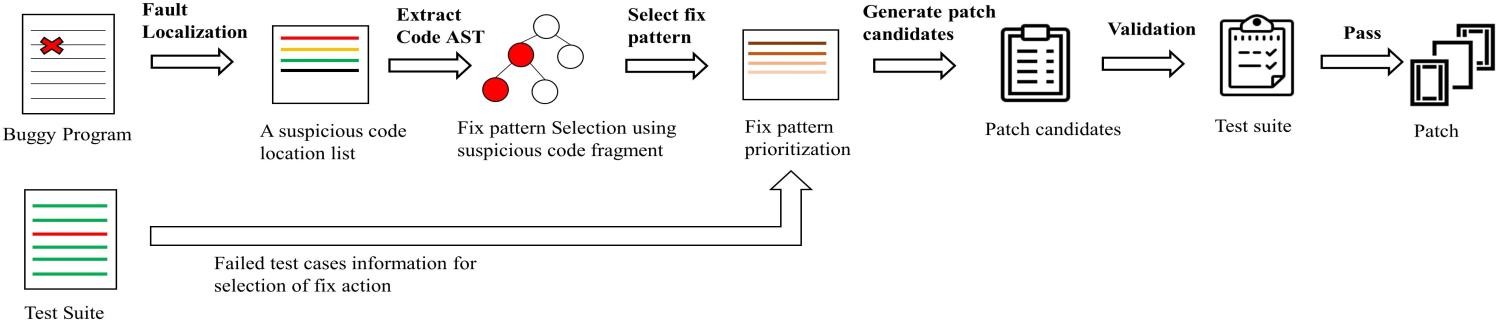


Fig. 1. Workflow of Fix director based APR first identifies fault locations, i.e., statements and

uses fix pattern previously extracted to generate patch candidates. Finally, validate the patch candidate checking

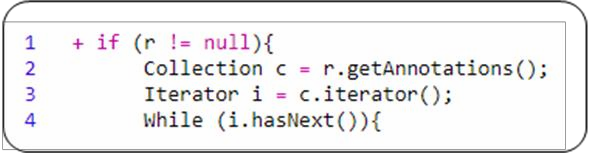
1. APPROACH

passes all given test cases. This process reduces the costs and increase possibilities to find correct patches more than traditional approach, i.e., search-based APR.

1. *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 with 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 select fix pattern with their own approaches with fault statement information, but there are no 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 this information, Developers repair the program. If APR techniques can get information from failed test cases and utilize it, it will be easy to decide which 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 APR techniques modify the program according to the selected fix pattern without specifying which problem to fix. This leads to 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.



**Fig. 2. The faulty code from Chart-4**

Figure 1 shows an overview flow of our approach. Fault localization step is necessary. It identifies a list of suspicious code locations where we apply the fix action. In our experiment, we used the perfect localization which means 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 are generated for each suspicious statement.

Since classifying fix patterns works are similar with classifying the behavior of developers who fix bugs, consideration for bug type needs to be given to 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 that 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 patterns generate at least one to dozens of candidate patches. These pattern prioritization tasks are eventually designed to generate the right candidate patches faster and follow the process of real developers fixing bug fixes. Because selecting fix patterns is the same process as choosing what kind of fix action developers will take, utilizing error messages as a basis for selecting fix pattern can be reasonable decision making.

TABLE I. FIX PATTERN RANK

1. EVALUATION

|  |  |  |
| --- | --- | --- |
| **Budged** | **Pattern rank** | |
| ***without prioritization*** | ***with Prioritization*** |
| Chart\_1 | 1 | 1 |
| Chart\_4 | 3 | 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 |
| Time\_7 | 2 | 1 |
| Time\_19 | 1 | 1 |

* 1. *Experiment setup*

For evaluating our approach, we selected 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 of fault localizaion noise, we implemented experiment with given buggy statement location. We implemented experiments in two ways. First, we reproduced TBar, currently known as the best effective APR technique in the world. When fixing each bug, we recorded the rank on which fix pattern we found the correct patch. The second experiment was to use information from the failed test. And we prioritized a more appropriate fix pattern more related with case of each bug. And then we recorded the pattern rank as in the first experiment.

* 1. *Research Questions and Results*

RQ1. How effective is the fix pattern prioritization?

We implemented two experiments. One was without fix pattern prioritization and the other was with fix pattern prioritization Table 1 shows the results of two cases showing the rank of the pattern in which the first correct patch was found. What 1 means is that the correct patch was found in the first fix pattern. 6 means that it failed for the 5 fix patterns applied and succeed with the 6th fix pattern. In case of 15 bugs, pattern rankings rose. Since multiple candidate patches are created in a single fix pattern, change in the fix pattern ranking 1 may be a more change in patch candidate rankings.

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

We collected 8 failed messages type. For each type of errors, we defined suitable fix patterns and applied to the prioritization. This is similar with developers bug fixing process when they manually fix bugs. Information on what type of errors were the basis for which fix action should be chosen.

RQ3. Failed test cases information covers all cases?

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

1. 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, which is still less efficient than developers manually fixing bugs. Because most APR technologies focused on bug location to fix the bugs without consideration as to why the bugs occurred, we try to fix action based on why the bug occurred in this study. 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 occurred.

1. FUTURE WORK

We conducted fix pattern prioritization for eight error message types. However, more error messages need to be classified because more diverse error messages exist 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, additional information on what causes bugs is needed.

Thus, future research will be conducted with the aim of complementing the above limitations.

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

2013.

1. 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.
2. Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large- scale experiment on the defects4j dataset. Empirical Software Engineering, pages
3. 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
4. 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.
5. 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.
6. 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
7. 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.
8. 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.
9. 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)

1. 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.
2. 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.
3. 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.
4. Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In ICSE.
5. 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.
6. 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.
7. 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.
8. 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
9. 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. ISBN 978-1-4503-6224-5. doi: 10.1145/3293882.3330577.
10. 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.
11. 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.
12. 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.
13. 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
14. 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.
15. Kai Pan, Sunghun Kim, and E James Whitehead. 2009. Toward an understanding of bug fix patterns. Empirical Software Engineering 14, 3 (2009), 286–315.
16. Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2017. An empirical analysis of the influence of fault space on search- based automated program repair. arXiv preprint arXiv:1707.05172 (2017).
17. 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.
18. 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)

⚫ **The below form will not be published, but it is necessary to help with the review process.**

**Authors’ Background**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | Email | Position  (Prof , Assoc. Prof. etc.) | **Research Field** | H**omepage URL** |
| Ju Hyoung Kim | [**Kjhkjh612@skku.edu**](mailto:Kjhkjh612@skku.edu) | **Student (master degree candidate)** | **Software engineering** |  |
| EunSeok Lee | [**leees@skku.edu**](mailto:leees@skku.edu) | **Prof** | **Software engineering** | **Selab.skku.ac.kr** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |