HKUST SPD - INSTITUTIONAL REPOSITORY

Title

Authors

Source

Version

DOI

Publisher

Copyright

This version is available at HKUST SPD - Institutional Repository (<https://repository.ust.hk/ir>)

If it is the author's pre-published version, changes introduced as a result of publishing processes such as copy-editing and formatting may not be reflected in this document. For a definitive version of this work, please refer to the published version.

Boosting Automated Program Repair with Bug-Inducing Commits

Ming Wen

mwenaa@hust.edu.cn

School of Cyber Science and Engineering, Huazhong University of Science and Technology
Wuhan, China

Yepang Liu

liuyp1@sustech.edu.cn

Department of Computer Science and Engineering, Southern University of Science and Technology
Shenzhen, China

Shing-Chi Cheung

scc@cse.ust.hk

Department of Computer Science and Engineering, the Hong Kong University of Science and Technology
Hong Kong, China

ABSTRACT

The search space explosion problem is a long-standing challenge for search-based automated program repair (APR). The *operation space*, which defines how to select appropriate mutation operators, and the *ingredient space*, which defines how to select appropriate code elements as fixing ingredients, are two major factors that determine the search space. Conventional approaches mainly devise fixing strategies via learning from frequent fixing patterns based on substantial patches collected from open-source projects. In this paper, we propose a new direction for search-based APR, that is to repair a bug via *learning from how the bug was introduced* instead of *learning from how other bugs are frequently fixed*. Our empirical study reveals that substantial mutation operators and fixing ingredients required to fix a bug can be inferred from the commit that introduced the bug. Based on the findings of our empirical study, we devised a preliminary fixing strategy based on bug-inducing commits, which is able to repair 8 new bugs that cannot be repaired by the state-of-the-art techniques. Such results demonstrate that our proposed new idea for searched-based APR is promising.

CCS CONCEPTS

• **Software and its engineering** → **Error handling and recovery**; *Software testing and debugging*.

KEYWORDS

Automated Program Repair, Bug-Inducing Commits

ACM Reference Format:

Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2020. Boosting Automated Program Repair with Bug-Inducing Commits. In *New Ideas and Emerging Results (ICSE-NIER'20)*, May 23–29, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3377816.3381743>

1 INTRODUCTION

Automated program repair (APR) has gained huge attention from both industry and academia recently. To date, a huge number of APR techniques have been proposed, among which search-based APR is

```
10549ef2 .../util/AuthUrlClient.java // Bug-Inducing Commit
-   stringBuilder.append(value);
+   stringBuilder.append(URLEncoder.encode(value,"UTF-8"));

b5524c90 .../service/DummyLogStreamingServlet.java // Bug-Fixing Commit
-   lastQueryString = request.getQueryString();
+   lastQueryString = URLDecoder.decode(request.getQueryString(),"UTF-8");
```

Figure 1: The Bug-Inducing and Fixing Commit of OOZIE-2320

a major category (e.g., [1, 7, 9, 14]). Although their effectiveness has been well demonstrated, limitations still exist which hinder their deployment in practice. One major and long-standing limitation of search-based APR is the space explosion problem [4, 15]. The search space of search-based APR is collectively attributed by the *fault space*, *operation space* and *ingredient space* [14]. The fault space is a set of suspicious code elements, and conventional APR techniques usually set it using existing fault localization techniques [14]. Conventional approaches set the operation space (i.e., mutation operators) via a predefined set of templates, or learning frequent fixing patterns from substantial real patches collected from fixing histories or open-source projects. Existing techniques search for appropriate fixing ingredients, from the buggy program itself, other programs of the project, or other applications. Intuitively, the wider range to search for fixing ingredients, the more likely that they can find the correct one. However, it will significantly aggravate the search space explosion problem.

To overcome the aforementioned limitation, we approach the automated program repair problem by taking a new perspective from the version histories of software. Software evolves by means of code changes committed to fix bugs, introduce new features, refactor existing code and so on. Such committed code changes, however, can also induce new bugs. Commits that induce software bugs are known as *bug-inducing commits*, which embed important contextual information for bug diagnosis [15, 16]. The information includes when and how bugs were introduced, and who made those improper operations that introduced the bug. Such information can be leveraged to advance existing APR techniques.

Motivated by this, we propose to repair a bug, in this study, via *learning from how it was introduced* instead of *learning from how other bugs are frequently fixed*, which is adopted widely by conventional approaches. Specifically, we are interested in the following research question in this study:

Research Question: *Can we search for mutation operators and fixing ingredients from a project's version histories, especially from the bug-inducing commits for search-based APR?*

Table 1: Mutation Operators and Fixing Ingredients of OOZIE-2320

Commit	Mutation Operator	Fixing Ingredient
Inducing	Insert, Method_Invocation	URLEncoder.encode
	Insert, String_Literal	"UTF-8"
	Move, Simple_Name	value
Fixing	Insert, Method_Invocation	URLEncoder.decode
	Insert, String_Literal	"UTF-8"
	Move, Simple_Name	request.getQueryString()

We use an example as shown in Figure 1 to illustrate our intuition. Figure 1 shows the bug-inducing commit and the associated bug-fixing commit for a real bug OOZIE-2320.¹ The bug was introduced by commit 10549ef2 since it enforced the encoding format to UTF-8 but forgot to make necessary changes at other places (e.g., source file DummyLogStreamingServlet.java in the example shown in Figure 1). To fix this bug, three mutation operators together with the associated fixing ingredients are required to be applied as shown in Table 1. As shown in this table, all the three mutation operators can be inferred from the associated bug-inducing commit. As for the fixing ingredients, "UTF-8" can be directly inferred while URLEncoder.decode is highly correlated with the one URLEncoder.encode involved in the bug-inducing commit. As for request.getQueryString(), it can be inferred from the buggy program itself. This example demonstrates that if the information of bug-inducing commit is available, users can be better guided to search for the appropriate mutation operators and the correct fixing ingredients to repair a bug.

Inspired by this example, we first performed an empirical study on the Defects4J [2] benchmark. Specifically, we observe that for 63.6% of the mutation operators applied to fix a bug, they can be directly inferred from the associated bug-inducing commits. Such a ratio could be enhanced to 69.3% on average if we also consider those inverse mutation operators extracted from bug-inducing commits. As for fixing ingredients, we observe that substantial of them can be inferred from the bug-inducing commits, especially for those with a few tokens. Besides, they are mainly involved in the inverse actions when leveraged to repair a bug compared with those actions that introduced the bug. Based on such findings, we derived a preliminary fixing strategy to repair bugs based on the information of bug-inducing commits. The results show that it is able to fix 8 new bugs in the Defects4J [2] that cannot be correctly repaired by the existing state-of-the-art techniques. Such findings and results demonstrate that our proposed new direction for APR is promising.

The rest of the paper is structured as follows. Section 2 presents the empirical evidence. Section 3 presents our preliminary approach together with the evaluation results. We discuss two important next steps in Section 4 and Section 5 concludes this work.

2 EMPIRICAL EVIDENCE

With the aim to answer the question raised in Section 1, we first conducted an empirical study to understand the ratio of mutation operators and fixing ingredients applied to fix a bug that could be inferred from its associated bug-inducing commits. In the subsequent sections, we first introduce the dataset used in this study and then present our empirical findings.

¹<https://issues.apache.org/jira/browse/OOZIE-2320>

Figure 2: The Ratio of Mutation Operators that Could be Inferred from the Bug-Inducing Commits

2.1 Dataset Collection

In this study, we leverage the DEFECTS4J [2] benchmark since it has been widely adopted in fault localization and automated program repair researches (e.g., [8, 10, 12, 14, 15]). A recent study [16] has identified the bug-inducing commits for 91 bugs in this benchmark via software testing, and the information of which is shown in Table 2. Our empirical study is carried out based on these bugs. Specifically, we first extract the mutation operators and fixing ingredients involved in both the bug-inducing commit and bug-fixing commit for each bug, and then compare their similarities.

Table 2: Subjects from the Defects4J Dataset

Subject	#Recovered	#Total	KLOC	Test KLOC	#Test Cases
Commons Lang	9	65	22	6	2,245
JFree Chart	11	26	96	50	2,205
Commons Math	24	106	85	19	3,602
Joda- Time	6	27	28	53	4,130
Closure Compiler	45	133	147	104	7,929
Total	95	357	378	232	20,111

2.2 Inferring Mutation Operators

An atomic mutation operator can be represented as a pair of (Operation, Node) [7, 14]. There are four kinds of Operation, which are Change, Add, Delete and Move. The Node represents the type of the changed AST node, the examples of which are Method_Invocation and Simple_Name as shown in Table 1. Figure 2 shows that on average, 63.6% (i.e., ranging from 51.2% to 77.2% for different projects) of the mutation operators used to fix a bug can be directly inferred from the associated bug-inducing commits. Since it is also common to revert those changes that introduced a bug with the aim to fix it [15, 16], we also investigate whether such ratios can be enhanced if we reverse those mutation operators that introduced the bug. Specifically, the inverse of (Insert, Simple_Name) is (Delete, Simple_Name) and the inverse of (Delete, Simple_Name) is (Insert, Simple_Name) accordingly. As for operations of Update and Move, the inverse actions are themselves. As we can see in Figure 2, after reversing those mutation operators applied in the bug-inducing commits, the ratio can be enhanced to 69.3% on average (i.e., ranging from 60.2% to 88.0% over different projects). Based on these results, we can distill the following finding:

Finding #1. The mutation operators to fix a bug can be inferred, either directly or inversely, from the associated bug-inducing commits for a substantial number of cases.

2.3 Inferring Fixing Ingredients

Inferring appropriate fixing ingredients is critical in automated program repair. Some studies infer them from the buggy class itself (e.g., [14]), some infer from the same application (e.g., [6]), or from other projects (e.g., [13]). The wider range to search for fix ingredients, the more likely that we can find the correct one. However, it will increase the search space exponentially, and thus hinder the APR technique to generate a correct patch within the given time budget. In this study, we propose to look for appropriate fixing ingredients with respect to version histories, especially the bug-inducing commits. Specifically, we compute the ratio of fixing ingredients that can be inferred from the bug-inducing commits. Fixing ingredients vary in their sizes, some of which might be composed of many tokens. In this investigation, we first perform lexical analysis for each ingredient and create a vector of lexical tokens following the practice of existing studies [9, 15]. The identifiers that are composed of multiple tokens (e.g., `getQueryString`) are split into individual tokens (e.g., `get`, `Query` and `String`). We then use the length of the token vector to denote the length of the fixing ingredient. Figure 3 shows the empirical results. Nearly 60.0% of the fixing ingredients of length 1 can be directly inferred from the bug-inducing commits, that ratio is 35.4% for ingredients of length 2, and 30.5% for length 3. For those ingredients that are large in sizes (e.g., length 13), we can still find a nonnegligible portion of them from the bug-inducing commits (e.g., 25.0%).

For those fixing ingredients that can be inferred from the bug-inducing commits, we further investigated whether they are involved in the same actions or the inverse actions compared with the actions performed to fix the bug. For instance, if the inducing commit has *inserted* a specific *ingredient*, we are curious whether it is more likely to perform the same action (i.e., *insert*) on the ingredient when fixing the bug, or to perform the inverse action (i.e., *delete*). As shown in Figure 3, the majority of the ingredients that can be inferred from the bug-inducing commits are involved in the inverse actions when fixing the bug compared with the actions that introduced the bug. For instance, for ingredients of length 2, 74.4% of them are involved in the inverse actions while this ratio is only 43.0% for the same actions. Note that the sum of these two values (74.4% + 43.0%) can be greater than 1 since there might be multiple places to find the ingredients, and thus the ingredients might be involved in both the same action and the inverse action. Based on these results, we can distill the following finding:

Finding #2. Substantial fixing ingredients can be inferred from the bug-inducing commits, especially for those with a few tokens. Besides, they are mainly involved in the inverse actions when leveraged to fix a bug compared with the ones that introduced it.

3 PRELIMINARY APPROACH

Our Finding #2 indicates that substantial fixing ingredients leveraged to fix bugs can be inferred from the bug-inducing commits. In particular, they are mainly leveraged by reverting those change actions performed in the corresponding bug-inducing commits. This finding can guide the search of mutation operators and fixing ingredients in designing APR techniques. To demonstrate the feasibility, we performed the following study.

Figure 3: The Ratio of Fixing Ingredients that Could be Inferred from the Bug-Inducing Commits

Methodology: We adopt a simple strategy based on Finding #1 and Finding #2 to repair those bugs as shown in Table 2 based on the bug-inducing commits. The strategy works in two steps. First, we revert the whole changes together with the associated ingredients performed in the bug-inducing commits. Second, if the first step cannot repair the bug, we try to revert a subset of the changes, together with the associated ingredients, performed in the bug-inducing commits. After applying the two steps, a bug can be: 1) *not repaired*; 2) *fully repaired*, in which case all the change actions required to fix the bug are performed; and 3) *partially repaired*, in which case some changes required to fix the bug are performed but fixing the bug still requires applying other mutation operators and fixing ingredients.

Results: Table 3 shows the results. It includes the information of the identified bug-inducing commit (BIC), the strategy adopted (i.e., fully reverting BIC or partially reverting it) and the result (i.e., whether the bug is fully repaired or partially repaired). In total, 11 bugs can be fully repaired and one bug can be partially repaired using our strategy. Among them, 8 bugs have never been reported to be repaired by the state-of-the-art APR techniques [1, 3, 11, 14, 17]. We chose these five APR techniques, since they are the most effective ones according to the recent study [5]. We further investigated those 8 cases, and found several reasons. First, multiple mutation operators are required to fix the majority of those bugs while existing studies only support applying a single mutation operator in their design (e.g., CapGen [14]). Second, for certain bugs, the correct fixing ingredients exist only in historical versions (e.g., bug-inducing commits) instead of the current buggy version while most of the existing techniques search for fixing ingredients only in the current buggy version [14]. The above results indicate that the direction proposed in this study, that is to leverage bug-inducing commits to repair bugs automatically, is promising.

Table 3: Bugs that Are Repaired by Our Strategy

Project	Bid	BIC	Strategy	Result
Closure	31	#a4c526d	partially revert	fully repaired
Closure	61	#e9a1cc9	fully revert	fully repaired
Closure	62 ♣♣	#22a1fcd	fully revert	fully repaired
Closure	66	#70f817a	partially revert	fully repaired
Closure	85	#2581851	partially revert	partially repaired
Closure	92	#5a3b4cb	fully revert	fully repaired
Closure	113	#87071dd	fully revert	fully repaired
Closure	115 ♠⊙	#b80dfd3	fully revert	fully repaired
Closure	120	#000ea0f	partially revert	fully repaired
Math	46	#330f3fe	partially revert	fully repaired
Math	50 ♠⊙⊙	#000ea0f	partially revert	fully repaired
Math	53 ♡♠	#6ef3b29	partially revert	fully repaired

BIC denotes *bug-inducing commit*. Symbol ⊙ denotes the bug can be repaired by CapGen [14], ♠ by SimFix [1], ♡ by ELIXIR [11], ♠ by FixMiner [3] and ⊙ by ssFix [17]

4 WHAT'S NEXT

We identify two major important items of future work along this direction for search-based APR.

4.1 Identifying Bug-Inducing Commits

The information of bug-inducing commits plays a critical role in the proposed idea in this study. However, identifying bug-inducing commits precisely and effectively is challenging. Existing studies perform binary search for bug-inducing commits via executing bug-revealing tests on historical versions [16]. However, this approach can only identify 25.5% of the bug-inducing commits precisely on the Defects4J benchmark as shown in Table 2. Such imprecision commonly occurs in the situations when the bug-revealing tests depend on code elements missing from or incompatible with historical versions. Therefore, one of our important next steps is to propose new techniques with the aim to identify bug-inducing commits precisely and effectively for a larger number of cases. Specifically, we plan to analyze the effect of a commit via change impact analysis, and compare it with the test coverage of the bug-revealing tests to prioritize suspicious commits. Besides, for those cases whose bug-inducing commits cannot be precisely identified through testing, we plan to investigate whether automated program repair can be better guided via learning from a set of potential commits that are correlated with the bug.

4.2 Inferring Sophisticated Rules

In the preliminary experiments of this study, we simply reverted those mutation operators and fixing ingredients involved in the bug-inducing commits with the aim to repair bugs. However, this strategy can only repair a small portion of bugs as indicated by the results in Section 3. As a result, more sophisticated rules are desired to infer mutation operators and fixing ingredients based on the information of bug-inducing commits. For instance, for mutation operators, it has been observed that operator `<Delete, Expression_Statement>` in bug-inducing commits is significantly correlated with the one `<Insert, Field_Declaration>` in bug-fixing commits [16]. This suggests that if a bug is introduced via deleting an *expression statement*, it is more likely to be fixed via inserting a *field declaration*. Such correlations can guide us to derive more effective rules from bug-inducing commits to design automated repair techniques. As for fixing ingredients, although some of them cannot be directly inferred from the bug-inducing commits, they are highly correlated. Therefore, program analysis can be incorporated to infer fixing ingredients more precisely. For instance, in the example as shown in Figure 1, we can extract those similar method calls to infer ingredient `decode`. We leave such explorations as our important future work.

5 CONCLUSION

In this paper, we propose to perform automated program repair in terms of a new direction, that is to infer mutation operators and fixing ingredients from the perspective of a bug's inducing commits. Mutation operators and fixing ingredients such as inferred are specific to the target bug to be repaired. On the contrary, conventional approaches mainly infer general mutation operators from substantial existing fixing patches collected from open-source projects. Our empirical study confirms our intuition, and demonstrates that such

a direction is promising. Based on the findings of our empirical study, we designed a preliminary fixing strategy, which can repair 8 new bugs that cannot be correctly repaired by the existing state-of-the-art techniques. We also pointed out important future research directions to better utilize the information of bug-inducing commits for better automated program repair.

ACKNOWLEDGMENTS

This project is sponsored by the National Natural Science Foundation of China (No. 61932021 and 61802164) and the MSRA Collaborative Research Grant.

REFERENCES

- [1] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 298–309.
- [2] 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 2014 International Symposium on Software Testing and Analysis*. 437–440.
- [3] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. Fixminer: Mining relevant fix patterns for automated program repair. *arXiv preprint arXiv:1810.01791* (2018).
- [4] Claire Le Goues, Stephanie Forrest, and Westley Weimer. 2013. Current challenges in automatic software repair. *Software quality journal* 21, 3 (2013), 421–443.
- [5] 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 *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 102–113.
- [6] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 166–178.
- [7] 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.
- [8] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 441–444.
- [9] Matias Martinez, Westley Weimer, and Martin Monperrus. 2014. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 492–495.
- [10] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 609–620.
- [11] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 648–659.
- [12] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.
- [13] Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. 2015. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 43–54.
- [14] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1–11.
- [15] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 262–273.
- [16] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 326–337.
- [17] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–670.