Tidy Up Your Source Code! Eliminating Wasteful Statements in Automatically Repaired Source Code

Takumi Iwase(✉), Shinsuke Matsumoto, and Shinji Kusumoto

Graduate School of Information Science and Technology,
Osaka University, Osaka, Japan
{tk-iwase,shinsuke,kusumoto}@ist.osaka-u.ac.jp

Abstract. Automated program repair (APR) is a concept of automatically fixing bugs in source code to free developers from the burden of debugging. One of the issues facing search-based APR is that repaired code contains wasteful or meaningless statements that do not affect external behavior. This paper proposes a concept named *source code tidying* that eliminates wasteful statements in source code repaired by search-based APR. Our proposed method applies pre-defined tidying rules to repaired code and evaluates the effect of tidying using source code metrics such as lines of code. By repeating this process based on a genetic algorithm, unnatural and full of wasteful source code is gradually brought close to natural with preserving its behavior. Our method will be involved in a process of APR by improving the readability of repaired code.

Keywords: Automated program repair · Source code tidying · Wasteful statements · Dead code · Refactoring

1 Introduction

Automated program repair (APR) is a concept of automatically fixing bugs in source code to free developers from the burden of debugging [5]. APR can be broadly classified into search-based [6] and semantics-based [9] approaches. This paper focuses on genetic algorithm-based APR (GA-APR), one search-based APR that introduces bio-inspired evolution into program repair. GA-APR takes as input source code containing one or more bugs and test cases. GA-APR repeatedly applies tiny modifications to the buggy code until all test cases pass. While the semantics-based approach is limited to a specific type of bug, such as conditional bug [12], the search-based approach has the significant advantage of generality in that it can theoretically fix any kind of bug.

One of the issues facing GA-APR is that repaired code contains wasteful or meaningless statements that do not affect external behavior. Usually, GA-APR repeatedly applies predefined modifications without considering semantic information. Typical modifications include insertion/deletion/reuse of AST nodes

D. Taibi et al. (Eds.): PROFES 2022, LNCS 13709, pp. 544–550, 2022.
https://doi.org/10.1007/978-3-031-21388-5_40

[6], insertion/deletion of method calls [2], and modification of variable names or operators [1]. These blind and random modifications lead to a problem that repaired source code tends to be far from the source code written by developers. For example, repetitive insertion of AST nodes will generate wasteful statements such as "`n++; n--;`", which negate each other, or "`n++; n=10;`", in which the former statement is overwritten. There is also a case where only an empty block "`{}`" is left due to repetitive deletion. The number of applied modifications will increase if a bug is difficult to repair. Many modifications make repaired code full of wasteful statements. As a result, overall repair performance (i.e., search performance) will gradually decrease with increasing the generations because wasteful statements affect the performance of compilation and test execution.

This paper proposes a concept named *source code tidying* that eliminates wasteful statements in source code repaired by GA-APR. We define wasteful statements as executable statements that do not affect external behavior. This definition includes not only dead code [4,10], which is a well-known concept of unused and unreachable code, but also used and reachable but unnecessary. Our proposed method applies predefined tidying rules to repaired code and evaluates the effect of tidying using source code metrics such as lines of code. By repeating this process based on a genetic algorithm, unnatural and full of wasteful source code is gradually brought close to natural with preserving its behavior. Our method will be involved in a process of GA-APR by improving the readability of repaired code.

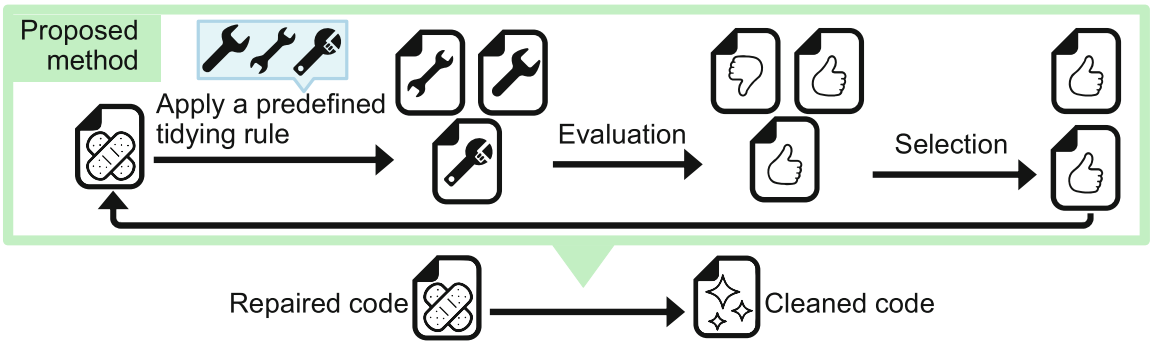

Fig. 1. Overview of proposed method

2 Proposed Method

2.1 Overview

The purpose of the proposed method is to *tidy* source code that contains wasteful statements. We define wasteful statements as executable statements that do not affect external behavior and *tidy* as eliminating these wasteful statements. Figure 1 shows an overview of the proposed method. The input is repaired source code by GA-APR, and the output is tidied source code which is same behavior

as the input. The proposed method consists of three iterative processes using genetic algorithm: tidying, evaluation, and selection. First, the source code is partially tidied by a randomly selected rule from predefined tidying rules. At this time, the decision on which rule to use is made several times. This results in multiple tidied source codes from a single source code. Next, evaluate each partially tidied source code. As fitness, we use metrics such as lines of code and cyclomatic complexity. Then, *good* source codes are selected to the next tidying based on fitness. If the fitness does not improve after repeating these processes, the iteration finishes and the proposed method outputs source code with the best fitness.

The proposed method has two features: tidying rules can be added, and the source code is tidied based on GA. Even if the proposed method fails to tidy some source codes, the proposed method will be able to tidy them by adding rules. GA-based tidying enables natural tidying as humans do.

2.2 Tidying Rules

Table 1 shows the tidying rules adopted in this paper. If rule affects the behavior of source codes when applying, it does not apply. For example, the swap rule for D1 in Table 1 does not apply to "`n++; m=n;`". Tidying rules are broadly classified into two. One is "direct rules", which directly eliminate wasteful statements. The other is "detour rules", which add or swap statements in the opposite direction to wasteful statements elimination. Wasteful statements in repaired source codes may not be adjacent. Direct rules eliminate adjacent wasteful statements and cannot eliminate nonadjacent wasteful statements. Therefore, we adopted detour rules in the role of gathering scattered wasteful statements. Detour rules are a major difference from related works, and we believe it works effectively for tidying of repaired source codes.

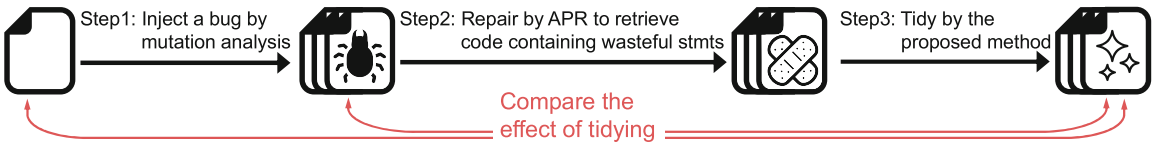

Fig. 2. Overview of experiment

3 Preliminary Experiment

3.1 Overview

We conduct a preliminary experiment using a programming contest as the subject. Figure 2 shows an overview of the experiment. This experiment consists of three steps. First, we inject bug into bug-free source codes (S_{origin}) and obtain

bug-injected source codes ($S_{mutated}$). Next, we repair $S_{mutated}$ by APR and obtain bug-repaired source codes ($S_{repaired}$). Finally, we tidy $S_{repaired}$ by the proposed method and obtain tidied source codes (S_{tidied}). We compare S_{tidied} with S_{origin} and $S_{repaired}$ to confirm effectiveness of tidying.

3.2 Experimental Procedure

Table 2 shows a list of the experimental settings and each step is described below.

Step1: Bug injection. S_{origin} are correct answers for twenty 100-point tasks of the past AtCoder Beginner Contest (ABC), held at AtCoder[1]. We inject bug into these correct answers. Mutation analysis [8] is used for the bug injection. In

Table 1. Tidying rules (Rn: direct rule, Dn: detour rule)

ID	Rule	Before	After
R1	Eliminate unary operator stmts that negate each other	`n++;` `n--;`	
R2	Eliminate overwritten stmts	`n++;` `n=1;`	`n=1;`
R3	Eliminate a block without having control stmt	`{ n++; }`	`n++;`

R4	Eliminate an empty block stmt	`n++;`
		`{ }`
		`n--;`
			`n++;`
			`n--;`

R5	Omit a control stmt whose condition is always true or false	`if(true){`
		`n++;`
		`}`
			`n++;`

R6	Merge a duplicate return stmt	`if(m>0){`
		`n++;`
		`return n;`
		`}`
		`return n;`
			`if(m>0){`
			`n++;`
			`}`
			`return n;`

ID	Rule	Before	After
D1	Swap two stmts that have no order dependence	`n++;` `m--;`	`m--;` `n++;`

D2	Inline a stmt located below control stmt	`if(m>0){`
		`n--;`
		`}else{`
		`}`
		`n++;`
			`if(m>0){`
			`n--;`
			`n++;`
			`}else{`
			`n++;`
			`}`

D3	Inline a stmt located above control stmt that has no dependence on condition	`n++;`
		`if(m>0){`
		`n--;`
		`}else{`
		`}`
			`if(m>0){`
			`n++;`
			`n--;`
			`}else{`
			`n++;`
			`}`

D4	Copy return stmt located below control stmt	`if(m>0){`
		`n++;`
		`}`
		`return n;`
			`if(m>0){`
			`n++;`
			`return n;`
			`}`
			`return n;`

[1] https://atcoder.jp/.

Table 2. Experimental settings

Parameter	Setting
S_{origin}	ABC[a] 100-point tasks
Number of tasks	20
Applied mutations	5 operations (see Table 3)
Used APR tool	kGenProg [7]
Number of $S_{repaired}$	76
Fitness in prop. method	Lines of code

[a] https://atcoder.jp/

Table 3. Mutation operations

Operation	Before	After
Replace + and -	`n = a + b`	`n = a - b`
Replace * and /	`n = a * b`	`n = a/b`
Replace % to *	`n = a % b`	`n = a * b`
Negate condition	`if (n > 0)`	`if (n <= 0)`
Change boundary	`if (n > 0)`	`if (n >= 0)`

mutation analysis, a single line in source code is modified by mutation. Table 3 shows the mutations adopted in this experiment. There are multiple operators and conditions that can be modified in a source code. We use all candidates and one candidate is used to generate one $S_{mutated}$. Therefore, multiple $S_{mutated}$ are generated from one S_{origin}. A total of 76 $S_{mutated}$ are generated from 20 S_{origin}.

Step2: Apply APR. The APR tool to repair $S_{mutated}$ is kGenProg [7]. Source codes of programming contest are simple and unlikely to contain wasteful statements when repaired. This makes it difficult to confirm effectiveness of the proposed method. Therefore, we generated multiple bug-repaired source codes from a single $S_{mutated}$. This increases the probability of generating source code with a lot of wasteful statements. The most wasteful source code is selected as $S_{repaired}$ among these multiple bug-repaired source codes.

Step3: Apply proposed method. The proposed method is applied to 76 $S_{repaired}$. As mentioned earlier, source codes for programming contest are simple. With metrics other than the lines of code (LOC), it is difficult to make a difference before and after tidying. Therefore, we use LOC as fitness in the proposed method.

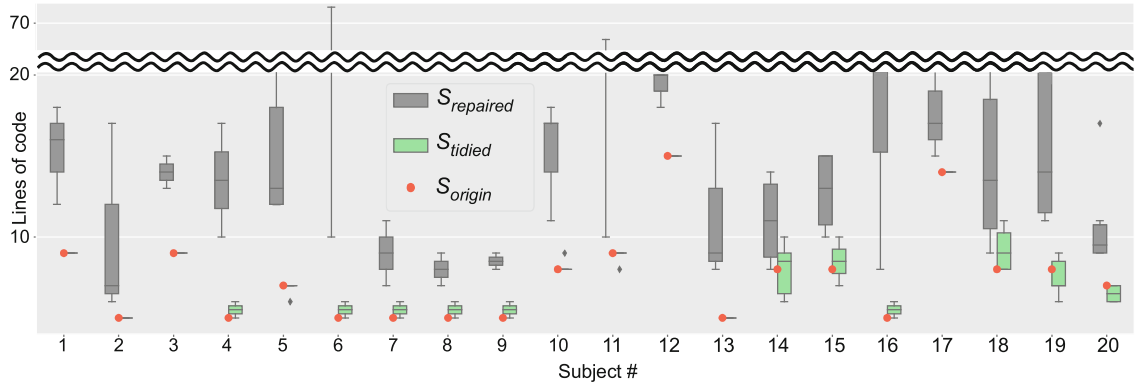

Fig. 3. Number of lines before and after tidying per subject

3.3 Results and Discussion

The Effect of Tidying: We confirm how many wasteful statements have been eliminated by the proposed method. Figure 3 shows LOC before and after tidying for each subject. The horizontal axis represents each subject, and the vertical axis represents LOC. The red dots represent LOC of S_{origin}. LOC decreased in all subjects, and we confirmed some source code tidied to LOC of S_{origin}. Next, manual check was carried out for each S_{tidied}. In 66 of the 76 source codes, wasteful statements were completely eliminated. Some source code had a different number of lines from S_{origin}, but no wasteful statements. This is because structure of conditional branches changed due to repair by APR. In the remaining 10 source codes, wasteful statements were not eliminated completely. The reason is the lack of tidying rules. By adding rules, we can eliminate wasteful statements in these source codes. From the above results, we consider that the proposed method can eliminate wasteful statements of source code.

4 Conclusions and Future Work

In this paper, we proposed the method to tidy APR-generated unnatural source code into natural source code. The proposed method tidies source code based on GA. We devised detour rules that do not directly eliminate wasteful statements. We conducted experiment using programming contest as subject. The obtained results showed that the proposed method could eliminate wasteful statements.

In future work, we expand tidying rules to improve the generality of the proposed method. This paper only focuses on fundamental arithmetic operators and basic control statements. Tidying rules for method invocation are necessary to apply our method to more practical source code. We consider that the key is checking the program dependences [3] and side-effect [11] of each statement.

Acknowledgments. This research was partially supported by JSPS KAKENHI Japan (Grant Number: JP21H04877, JP20H04166, JP21K18302, JP21K11829, JP21K11820, JP22H03567, and JP22K11985).

References

1. Assiri, F.Y., Bieman, J.M.: An assessment of the quality of automated program operator repair. In: Proceedings of International Conference on Software Testing, Verification and Validation, pp. 273–282 (2014)
2. Dallmeier, V., Zeller, A., Meyer, B.: Generating fixes from object behavior anomalies. In: Proceedings of International Conference on Automated Software Engineering, pp. 550–554 (2009)
3. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. on Program. Lang. Syst. **9**(3), 319–349 (1987)
4. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional, Boston (1999)

5. Gazzola, L., Micucci, D., Mariani, L.: Automatic software repair: a survey. IEEE Trans. on Softw. Eng. **45**(1), 34–67 (2019)
6. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: GenProg: a generic method for automatic software repair. IEEE Trans. on Softw. Eng. **38**(1), 54–72 (2012)
7. Higo, Y., et al.: kGenProg: a high-performance, high-extensibility and high-portability APR system. In: Proceedings of Asia-Pacific Software Engineering Conference, pp. 697–698 (2018)
8. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. on Softw. Eng. **37**(5), 649–678 (2010)
9. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: program repair via semantic analysis. In: Proceedings of International Conference on Software Engineering, pp. 772–781 (2013)
10. Romano, S., Vendome, C., Scanniello, G., Poshyvanyk, D.: A multi-study investigation into dead code. IEEE Trans. on Softw. Eng. **46**(1), 71–99 (2020)
11. Rountev, A.: Precise identification of side-effect-free methods in java. In: Proceedings of International Conference on Software Maintenance, pp. 82–91 (2004)
12. Xuan, J., Martinez, M., et al.: Nopol: automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. **43**(1), 34–55 (2017)