Anti-patterns for Java Automated Program Repair Tools

Yi Wu
11712738@mail.sustech.edu.cn
Southern University of Science and Technology

ABSTRACT

Prior study has identified common anti-patterns in automated repair for C programs. In this work, we study if the same problems exist in Java programs. We performed a manual inspection on the plausible patches generated by Java automated repair tools. We integrated anti-patterns in jGenProg2 and evaluated on Defects4J benchmark. The result shows that the average repair time is reduced by 22.6 % and the number of generated plausible patches is reduced from 67 to 29 for 14 bugs in total. Our study provided evidence about the effectiveness of applying anti-patterns in future Java automated repair tools.

ACM Reference Format:
Yi Wu. 2020. Anti-patterns for Java Automated Program Repair Tools. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3324884.3418919

1 PROBLEM AND MOTIVATION

In the test-driven automated program repair systems, a fix is validated if it can pass all the tests in the given test suite. Studies have revealed the insufficiency of using test suites as specification, which can lead to generating *plausible patches* (incorrect patches that may fail tests beyond a given test suite) by simply deleting program functionality [9]. Prior study [10] has identified common problems in automated repair for C programs and applied anti-patterns to improve the quality of program patches. Our goal is to study if the same anti-patterns will occur for Java program repair and further improve the Java automated repair tools.

2 BACKGROUND AND RELATED WORK

Many automated Java repair tools have been proposed [1–3, 6, 7, 11, 12]. SimFix [3] leverages existing patches from other projects and similar code snippets in the same project to generate patches. CapGen [11] uses context information of AST nodes to prioritize patches. LSRepair [6] utilizes fix ingredients at method declaration level. Astor [7] is an automated software repair framework for Java. It includes three implementations of repair approaches: jGenProg2, jKali, and jMutRepair.

Previous study [10] refers to anti-patterns as a set of forbidden transformations on programs. It defined a set of anti-patterns for C automated repair tools, which includes the following anti-patterns we use in our work: *Anti-append Trivial Condition*, *Anti-delete Control Statement*, and *Anti-append Early Exit.*

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3418919

3 APPROACH AND NOVELTY

3.1 Prevalence of Anti-patterns

We manually analyzed the plausible patches generated for Defects4J benchmark[4] by three Java automated repair tools: SimFix, CapGen, and LSRepair. Table 1 shows our manual inspection results. Our study indicates that the most common anti-patterns (*Delete if-statement*, *Delete loop*, and *Insert contradiction*) shown in the study [10] for C programs also occur in Java program repair tools despite recent advancement in Java repair techniques. The three tools exhibit different anti-patterns: *Anti-delete Control Statement* and *Anti-append Early Exit* only exist in LSRepair while CapGen only has *Anti-append Trivial Condition*. We think this difference is due to the distinct repair strategies the tools use: CapGen leverages fine-granularity fix ingredients in terms of AST nodes [11] while LSRepair searches for fix ingredients at the method level [6].

3.2 Examples of Anti-patterns

3.2.1 *Anti-append Trivial Condition.*

Insert contradiction. The patch is from SimFix for Defects4J Math80, where $k < 0$ is always false :

```
-   for (int k = 0; k < 4; k += step) {
+   for (int k = 0; k < 0; k += step) {
```

3.2.2 *Anti-delete Control Statement.*

Delete loop. The patch is from LSRepair for Lang52.

```
-    for (int i = 0; i < sz; i++) { ... }
```

3.2.3 *Anti-append Early Exit.*

Insert Early Return. The patch is from LSRepair for Defects4J Lang55. It deletes the whole method body and insert an exit.

```
    public void stop() {
-       ...
+       System.exit(0);
    }
```

3.3 Integrating Anti-pattern

As we derived anti-patterns on SimFix, CapGen, and LSRepair, we implemented anti-patterns on jGenProg2 to check if our proposed anti-patterns generalize across tools. jGenProg2 uses genetic programming to search fixes for programs. Algorithm 1 shows how we check if a program modification belongs to anti-patterns. We implemented the anti-patterns shown in Table 1: *Anti-delete Control Statement*, *Anti-append Early Exit*, and *Anti-append Trivial Condition*. Function *isControlStmt* checks if the removed statements are if-statements or loops. *isTrivialCondition* checks whether the conditions of if-statements and loops trivially evaluate to true/false. *isReturnBeforeStmt* and *isReturnNotLastStmt* checks if an early return is inserted. We use SPOON [8] library to implement the code.

Table 1: Proportion of Anti-patterns

	Total Number Of Plausible Patches Inspected	Anti-append Trivial Condition		Anti-delete Control Statement		Anti-append Early Exit
		Insert contradiction	Insert tautology	Delete if-statement	Delete loop	Insert early return/exit
SimFix	22	8.33%	0	0	0	0
CapGen	219	10.0%	1.83%	0	0	0
LSRepair	484	0	0	64.87%	19.63%	2.48%

Algorithm 2 shows how we integrated anti-patterns in jGenProg2. For each newly generated variant by function *createNewVariant*, we apply algorithm *isAntiPattern* to check if it is an anti-pattern. If the variant is an anti-pattern, we discard it. Otherwise, the function *processCreatedVariant* will compile then validate it using the test suite to see if it is a solution and it will be added to *temporalInstances* to be the parent variant for the next generation.

Algorithm 1: isAntiPattern

```
Input  :P: A program variant
Input  :Op: An operation on the program
Output :isAnti : True if P is an anti-pattern; False if not.
// Op.me: The modified element by the operation
isAnti ←False;
if Op is RemoveOp then
    isAnti ←isControlStmt(Op.me);
else if Op is ReplaceOp then
    isAnti ←isTrivialCondition(Op.me)
else if Op is InsertBeforeOp then
    isAnti ←isReturnBeforeStmt(Op.me);
else if Op is InsertAfterOp then
    isAnti ←isReturnNotLastStmt(Op.me);
return isAnti
```

Algorithm 2: jGenProg2 processGeneration

```
Input  :PV: A list of parent program variants
Input  :G: The number of generation
Output :findSolution : True if any validated patch generates.
findSolution ←False;
for each parentVariant in PV do
    newVariant ←createNewVariant(parentVariant, G);
    if ¬ isAntiPattern(newVariant) then
        temporalInstances.add(newVariant);
        if processCreatedVariant(newVariant) then
            findSolution ←True ;
        end if
    end if
end for
prepareNextGeneration(temporalInstances, G);
return findSolution;
```

4 RESULTS AND CONTRIBUTIONS

4.1 Experiment

We evaluate our anti-patterns in jGenProg2 using Defects4J benchmark. Our evaluation studies the following research questions: **RQ1**: How many plausible patches can anti-patterns eliminate? **RQ2**: How can anti-patterns affect correct patches? **RQ3**: How much can anti-patterns speed up the automated repair process?

We choose jGenProg2 because its prototype GenProg [5] is one of the most well-known repair tools. As our goal is to evaluate the effectiveness of anti-patterns in filtering plausible patches, we only focus on the 29 bugs where jGenProg2 generates patches [7] . We ran jGenProg2 on each bug up to 20 times with different seeds every time until it generated patches. We stopped and recorded the data the first run jGenProg2 produced any patches. In total, we successfully generated patches for 14 bugs.

4.2 Results and Discussions

Table 2 shows the experiment results. **RQ1**: In total, jGenProg2 generated 67 and 29 patches without and with anti-patterns respectively. The reduction is more than that of C programs for plausible patches generated by GenProg and SPR [10]. We think that this difference is due to that jGenProg2 implements lots of program modifications of *Anti-delete Control Statement* and *Anti-append Early Exit* as we observed when jGenProg2 is executing. Thus, our implemented anti-patterns could effectively prohibit these invalidated patches. **RQ2**: Anti-patterns can potentially affect correct patches (e.g., the correct repair requires deletion of control statements). In our test cases, the only correct patch generated for Math_70 is not affected. **RQ3**: The repair time for each bug was reduced with different degrees. The average time reduction is 22.6%. This result confirmed the effectiveness of anti-patterns in reducing the repair space by removing invalidated patches as observed in previous study [10].

Table 2: Evaluation Result

Bug ID	Total Time (s)		Number of Plausible Patches		Number of Correct Patches		Time Reduction
	Original	Anti-pattern	Original	Anti-pattern	Original	Anti-pattern	
Chart_1	222.30	202.54	2	0	0	0	8.9%
Chart_5	58.11	47.07	1	1	0	0	19.0%
Chart_7	55.32	46.45	0	0	0	0	16.0%
Chart_13	131.38	78.28	11	3	0	0	40.4%
Chart_15	267.48	260.00	1	1	0	0	2.8%
Chart_25	434.38	402.99	4	0	0	0	7.2%
Math_70	66.66	59.99	1	1	1	1	10.0%
Math_71	1121.80	762.26	1	1	0	0	32.1%
Math_73	44.17	33.62	1	1	0	0	23.9%
Math_80	1033.84	666.92	5	6	0	0	35.5%
Math_81	907.88	689.43	14	5	0	0	24.1%
Math_82	553.94	445.74	6	6	0	0	16.5%
Math_85	337.91	225.19	16	0	0	0	33.4%
Math_95	267.63	139.05	4	4	0	0	48.0%
Sum			67	29	1	1	
Average							22.6%

We also observed other potential anti-patterns when jGenProg2 is executing. For example, the plausible patch below is generated by jGenProg2 for Defects4J Math73. We would expect a single replacement of the assignment to the variable *delta* as consecutive assignments to the same variable is redundant. In the future, we can extend the anti-pattern set by including the potential anti-patterns.

```
1      delta = 0.5 * dx;
2  -   oldDelta = delta;
3  +   delta = (x0 - x1) / (1 - (y0 / y1));
```

4.3 Conclusion

In this study, we identified a set of anti-patterns in Java automated repair tools. We integrated anti-patterns in jGenProg2 and observed improvements in repair time and the number of produced plausible patches. This indicates the general applicability of anti-patterns in both Java and C programs. In the future, we intend to integrate anti-patterns into Java automated repair tools implemented by different approaches to optimize the repair process.

ACKNOWLEDGMENTS

I wish to thank Prof. Shin Hwei Tan for her guidance on this research project. This work was supported by the Department of Education of Guangdong Province, China (Grant No. SJJG201901).

REFERENCES

[1] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 307–318. https://doi.org/10.1109/ASE.2015.81

[2] Jiajun Jiang, Luyao Ren, Yingfei Xiong, and Lingming Zhang. 2019. Inferring Program Transformations from Singular Examples via Big Code. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 255–266. https://doi.org/10.1109/ASE.2019.00033

[3] 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* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[4] 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* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[5] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[6] Kui Liu, Anil Koyuncu, 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*. Nara, Japan.

[7] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 441–444. https://doi.org/10.1145/2931037.2948705

[8] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. SPOON: A Library for Implementing Analyses and Transformations of Java Source Code. *Softw. Pract. Exper.* 46, 9 (Sept. 2016), 1155–1179. https://doi.org/10.1002/spe.2346

[9] 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 *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 24–36. https://doi.org/10.1145/2771783.2771791

[10] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/2950290.2950295

[11] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

[12] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-Related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 660–670.