Final Report

Robin Zhang, lz2811@columbia.edu

Synopsis

Overview

Automatic Program Repair (APR) is massively adopted by the developers by automatically generating different patches so that the software could function without eliminating the expected features. But such repair has limitations: their patches is incorrect since the APR is not complete. One significant problem is patch overfitting, which means the APR program generates many patches that exceed the actual test suite. Thus, the usage of formal verification could be a possible way to mitigate such problems. Formal verification is known for its completeness and the adoption of static analysis, which is a great compensation for APR programs. Thus, I am proposing a tool/framework that evaluates the effectiveness of an APR tool for the developers by analyzing its possible of overfitting.

Novelty

This program is mainly focused on assessing the overfitting problem on APR tools working on Java programs. The Java developers could see the possibility of a certain APR tool, such as NOPOL and ARJA, overfitting their program using this framework, which takes different modules from the program and verifies the patches the APR tool generates. OpenJML will be used as the verifier of Java programs in this framework. And the testing examples and benchmark standards are from Defects4J and an open repository[1]. The repository provides 537 buggy sample programs with written formal specifications which can be used on different APR tools. They cover a large number of functionalities normally seen in common Java programs. And these sample programs could be extended to be tested on more Java APR tools used in the repository.

Value to the User Community

This project is mainly designed for developers with some independent projects which are written in Java and have needs to be auto repair. Or if the developers need to pick an appropriate APR tool to fix their bugs before they thoroughly look through the code and try to fix them, this framework would be an excellent help for choosing the APR tool. It might not be a very efficient way at the beginning of the development, but it could save times on using those APR tools especially if the developers pick the best suited one at the beginning. For Java programs, there are a lot of different APR tools, and it is hard for the developers to learn their pros and cons and decide one to use, so this framework is aimed at helping them finding the probably the best suited one for their own project by providing an assessment of the overfitting problem of each APR tools on the software program.

However, the current version of the framework does not achieve the proposed benefit yet.

[1] https://github.com/Amirfarhad-Nilizadeh/BuggyJavaJML

Research Questions

Methodology

The framework will take the program which is going to be tested on and arguments containing the name of the Java APR tools provided that the developer is willing to use as the input. Then it will compile the test program after it locates the jar file of one chosen APR tool. But here, to simulate the execution of APR patch generation and validation, for each APR tool, the rest arguments have been assigned by a default value for simplicity, which might be the reason some tests could not execute or keep reporting errors due to arguments misuse. For example, the usage of Nopol is limited to only minimal parameters: source code path, project classpath, lists of tests of the project, and solver path. The rest parameters such as mode and type are all set to default. Then the next step would take a long time to run, and it will report the validated patches and all other output from APR tools and the times one used. Logs will not be reported.

Then the OpenJML verifier would verify whether the solution from each APR tools could satisfy the specifications generated from the program and its requirements. But a major issue here is that the specification generation has not been incorporated into the entire framework yet, so some program will cause program to crash if the generator does not accept the program, the generator crashes, or the generated specification cannot be identified by OpenJML. Thus, here the BuggyJML dataset is used as replacement which would most likely solve the above problem. The parameters of running OpenJML are also being pre-selected. Then the SMT solver from OpenJML will run a static analysis trying to identify the AFL result of the output from each APR tools. It will report the number of not-repaired, validated-repair, verified-repair, and not-verified-repair for each selected APR tool. Lastly, the framework would do a simple analysis on these given data and provide a relatively acceptable suggestion on best performance APR tool which contains lowest percentage of overfitting patches of all validated patches.

Research Questions

1. *How efficient of the framework when assessing each APR tool? And how much time does it save comparing to the situation without using this framework?*
2. *How does the program complexity affect the APR tools? How does it affect the performance of this framework?*
3. *For what features or functions in a program most likely cause each APR tool overfitting problem?*

RQ1: APR Efficiency

How efficient of the framework when assessing each APR tool? And how much time does it save comparing to the situation without using this framework?

This framework is supposed to bring efficiency to developers who need to choose one APR tool for their program. If this framework is computationally hard itself, then value it provides is limited. This assessment also considers the situation if the developer picks the worst APR tool comparing to using the framework and choosing relatively the best one.

The simplest answer to this question is the framework fail to provide efficiency to the users. Given larger overhead, the framework still executes the key component of each selected APR tools, which seems to be inevitable. But these processes are the most time-consuming section of the tool itself, so the total number of time executing across different APR tools only skip through several parts that is not originally time consuming before we verify the patches. But we also added the part of verification, and they are not related to APR patch generation on the lower level there's no shortcut for it. Even though comparing to running APR tools verification is relatively fast, it still takes significant amount of time (> 9% of entire running time of the framework) that slow down the whole process. And an even worse new is that the time now does not include specification generation if we are using a completely unseen testing program. Those generation could also be time-consuming comparing to the final verification process, so the real running time of the framework could be even slow that what we have now. A typical 18 hours of running is not acceptable for a lot of users, since if they only pick one APR tool to run, it will take a few hours less. And if we add more APR tool to the test, the time will increase approximately linearly. This is a empirical conclusion by observing similar APR tool used together gives a sublinear time increase, but not similar APR tools together gives at least linear increase and some overhead. Even though it is acceptable for time complexity analysis, it won't be accepted by the users given the fact the most time used are multiplied, and more than an hour of time increase is noticeable by human and hurt their experience a lot.

One argument that might mitigate such low efficiency is that by analyzing the overfitting rate of some special part of the code, the user could see which APR tool performs better on what type of bugs to fix. If the future development involves multiple auto fix on part of the code which potentially contains similar bugs occurred before, the framework result would be helpful to predict the best performance APR tool for long term usage. And this step involves the idea of comparing new code or bugs with existing code in the library and calculate their similarities, this is out of the scope of this project. I have once thought about doing a code comparison to boost performance but didn't proceed due to lack of knowledge and time. But I think it could be a new approach to achieve a similar outcome as this framework but within a much shorter time frame.

RQ2: Program Complexity

How does the program complexity affect the APR tools? How does it affect the performance of this framework?

Different APR tools might suit different program, but for one program, does this tool still the best suited one to use to fix bugs as the program gets much bigger and complicated. Also, is the complexity the main factor of causing overfitting in APR? Or does the variety of different programs affect the most?

To simplify the study, I only calculated the time complexity that I have tested, so the number of examples is limited. Since the dataset only consists of functional programs, so this is not a hard step to compute. Besides a special case where the fix changes the program complexity, the rest tested program's time complexity seems to be no dramatic effect on the performance of each framework. For example, one program from BuggyJava JML "Binary Search" has time complexity of O(log(n)), and the complexity of another program "Bubble Sort" is O(n^2), but their running time are relatively close for Nopol, kGenProg, ARJA, and Cardumen. The average time difference is less than 5%, and there is even the case that "Binary Search" takes more time than "Bubble Sort" on Nopol.

However, ARJA and Cardumen actually produce a fix for "Binary Search Bug 9" which changes their complexity from O(log(n)) to O(n), and OpenJML successfully detect this problem and does not verify its correctness, so these patches are characterized as overfit patches.

Since the framework does not monitor the use of the space and I didn't calculate each programs space complexity, so I don't have any conclusion on either how the space complexity of the program impacts the performance of APR or how the program time complexity affect the maximum space used to run APR tools.

RQ3: Cause of Overfitting

For what features or functions in a program most likely cause each APR tool overfitting problem?

At the level of Java APR tools, it could be helpful to the developers if they have a guide or comparison summarization of those tools before choosing. What decides the overfitting results for each APR tool is a clear and precise assessment.

I have analyzed 64 out of 213 overfitting patches generated from the programs used on the framework, but most of them does not exhibit any distinct pattern of function or feature. Most of them are seems random and irrelevant. For example, in "Binary Search Bug 9", the patch that changes the function complexity actually changes the line of "*mid = low + (high - low) / 2;*" into "*mid = low + (low - low) / 2;*", which seems to be a random change on the variable and the validated patch does not seems to make sense by letting "*mid = low;*".

However, a more interesting result that we can observe is that which APR tool is more likely to generate overfitting patches. ARJA seems to be the most unreliable one with 78 unverified patches. Here I regard unverified patches as overfitting patches, since we won't distinguish the false negative on verification process and the number of those false negative cases are relatively small. And jKali only produce 1 overfitting patch on the programs I tested. But on the other hand, ARJA is more likely to produce validated patches comparing to jKali. And the overfitting rate (number of overfitting patches / number of validated patches) for ARJA is 30.83%, but the overfitting rate for jKali is 33.33%.

And ARJA mostly making overfitting patches on the program "Time", which produces 49 overfitting patches out of 53 validated patches. My conjecture of the poor performance of ARJA is it does not treated time as an irregular structured number instead of a more stable numerical number. And the test cases do not detect such problems for ARJA and let it slip through.

Deliverables

The code of the framework and dataset are included on this GitHub repository link: https://github.com/RobinZhang14ZLP/TSE6156FinalProjectAPR. And the README file also contains a short description of this project, a explanation of the repository structure, and also code, data ,and previous assignments PDF files.

The entire project is executed on my personal laptop. And I'll list a few key hardware parameters here: System: Windows 11; CPU: 16-core AMD Ryzen 7 5800U 1.9GHz; RAM: 16 GB. The result data might varies running on different device, also some data is not accurate due to the fact there are many other processes running at the same time.

Self-Evaluation

The first challenge that I had during these 2 months was setup every framework, APR tool, verifier, dataset, and benchmark. Even though Defects4J is not officially used in my project as a metric, there are a lot of the available APR related packages has dependency on it. There are a few integrated features from Defects4J are being used as a basic setup step in some APR tools, such as Nopol, and some of them require different environment settings. This is the reason why the APR tools that I selected in the framework is less than my original proposal, some settings contradict with each other, and I couldn't find a solution for both. TBar is a commonly used APR tool but I left it out for this project for the above reason.

Also, each APR tool has its own usage, and it takes time to learn all. Despite a few that are similar, Nopol, SimFix, and kGenProg are much different from the rest. ARJA, jKali, jMutRepair are all related to Astor, and it probably contains a similar implementation for their kernel. Nopol is more semantic-driven, while the other focus on the nature of generate-and-validation of test suites. And SimFix consists of a very large code mapping procedure during the generation of patches, which represents an essential overlap between program repair and code similarity detection. Synchronizing buggy codes, their testing suites, and specifications for each chosen APR tools took me a long time to complete, even though there's likely it will crash for some special cases listed in the dataset.

Since we already have the pre-generated specification for verification, I skip a big issue for integrating them together. But it still takes a long time to run each test and trial, and I even thought it went into some infinite loop and would never halt, so I kill the process several times until I learn relative normal time for APR tools.

I also have to change my plan and proposal several times when I learned more about APR and got to write actual code for building the framework. I was initially focused on how formal verification detect overfitting from APR tools, to how could formal verifier improve the performance of APR tools, till now analyzing the overfitting problem on APR tools with a verifier. I shifted a lot from the topic of my midterm paper, Formal Verification, to assessment of APR tools, especially the part involved patch overfitting. And I also have changed my framework design of how to get credible data to compare from using different APR tools. I have once tried to set up a machine learning engine to check the buggy program and testing code.

By doing the project, I've googled and read a lot of the research analysis of program repair and usage and comparison among those APR tools. I have basically no knowledge of program repair, except an overview of APR in the background section from a paper I read for my midterm paper on formal verification, and now even I won't say that I have gained proficiency on each of those tools, I believe I can quickly set it up and make it work if I'll need them someday in the future. The learning cost is quite big for a completely beginner. Also, I deep down into some cutting edge research on APR, and have learned what are the key impacts on their performance. I have also presented a paper on APR tools and flaky tests. Lastly, I have been through the entire process from a basic and vague idea to something that actually runs on a machine. How we can implement a simple idea into solid work is much harder than I thought, just like finding a correct and efficient algorithm to imitate some thinking process in human brain. There's a lot of detailed numbers to tune when which makes it tedious and time consuming, and it also shows how important that a team could really speedup the progress. Since I am working on my own, there's a lot of parallel tasks, i.e., configuring each APR tools, that could be dealt with at the same time if there are multiple people.

A few thoughts about the course

I think the logistic of the course efficiently provides us with both breath and depth on a few topics of the field of software engineering. Paper reading and presentation gives us opportunities to explore new technologies, cutting edge research on various topics. And a paper and a project allow us to dive in a certain topic that we like with both theory study and hand-on practice. I also find extremely important to learn how to read a research paper efficiently. So one suggestion that I have is maybe we can have one or two classes to overview each topics or subfields from software engineering and compare their general focuses. As a non-software-track student, I am not familiar with the ideas, and I was confused when we need to pick a topic for our midterm.

Resources

Testing dataset:

Buggy Java JML: https://github.com/Amirfarhad-Nilizadeh/BuggyJavaJML

Open JML: https://github.com/Amirfarhad-Nilizadeh/Java-JML

Benchmark framework:

Defects4J https://github.com/rjust/defects4j

APR Tools:

Nopol: https://github.com/SpoonLabs/nopol

SimFix: https://github.com/xgdsmileboy/SimFix

kGenProg: https://github.com/kusumotolab/kGenProg

jKali: https://github.com/SpoonLabs/astor/tree/master/src-jkali

jMutRepair: https://github.com/SpoonLabs/astor/tree/master/src-jmutrepair

ARJA: https://github.com/yyxhdy/arja

Cardumen: https://github.com/SpoonLabs/astor/tree/master/src-cardumen

Reference Paper:

Durieux, Thomas, et al. "Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts." *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019. https://dl-acm-org.ezproxy.cul.columbia.edu/doi/10.1145/3338906.3338911

A. Nilizadeh, G. T. Leavens, X. -B. D. Le, C. S. Păsăreanu and D. R. Cok, "Exploring True Test Overfitting in Dynamic Automated Program Repair using Formal Methods," 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST), 2021, pp. 229-240, doi: 10.1109/ICST49551.2021.00033. https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/document/9438573

Y. Qin, S. Wang, K. Liu, X. Mao and T. F. Bissyandé, "On the Impact of Flaky Tests in Automated Program Repair," 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2021, pp. 295-306, doi: 10.1109/SANER50967.2021.00035. https://ieeexplore-ieee-org.ezproxy.cul.columbia.edu/abstract/document/9425948