Automatic Software Repair

Paul-Philip Mosulet

April 2020

1 Introduction

Bugs are costly, either if developers spend time figuring out how to solve them, or if they just ignore them. The adoption of using SCA tools has been an ongoing process for the last decade. However, today the use of Static Code Analysis(SCA) tools is widely spread and common amongst larger tech-companies [1][2][3]. It is common to include these tools in the CI/CD pipelines, and without knowing it you might actually already be using one if you have a more sophisticated lint in your IDE. In their article from 2013 Johnson et al., interview 20 developers and ask them about their use of SCA tools, the results are then also confirmed in 2018, when Sadowski et al. mentions a couple of reasons why engineers may avoid fixing the issues displayed from the SCA tools in their article. Sadowski et al. list the following reasons: *not integrated* i.e. not included in the workflow or not automated, *not actionable*, *not trustworthy*, (does)*not manifest in practice* possible in theory but does not occur in practice, *to expensive to fix*, *warnings not understood* [1].

The first steps taken to improve these issues mentioned by Johnson et al. and Sadowski et al. were to improve the SCA tools themselves to better integrate with the developers workflow and make it easier to fix the issues [1][2][3]. Meanwhile, there has been ongoing research in the field of automated bug fixing based on the results from SCA [3][4]. It is hard to give an accurate estimate of the current state of the area in the industry given the nature of corporate secrecy. However, the industry also seems to be showing interest and seem to be in the starting blocks of adopting the field, with e.g. Facebooks Getafix as a front runner [5][6].

Automatic software repair is a discipline wherein the results of SCAs are combined with algorithms which suggest patches to the bugs found by the SCA. Since the area is still quite novel it is quite natural for scepticism from the community towards it. The current available solutions also vary in complexity and in implementation. But there are some general commonalities between the solutions. Firstly, there is some sort of fault detection component, e.g. some SCA tool or rule set. Secondly, there is a component which suggests one or multiple possible patches to the bug. Thirdly, the suggested patch is evaluated, then step 2 and 3 could be an iterative process until a desirable solution is

found. Finally, there is the step of patching the bug. This could be done in several ways: automatically in the pipeline, by creating a PR, as code-review on a PR or perhaps we could sometime have it just-in-time in our IDE working as a lint [4][7][8].

Based on this premise, the purpose of this essay will be to investigate the area of automatic software repair. The aim is to look into the theoretical background of automatic software repair and investigating the core concepts of the field. Furthermore, a couple of state of the art solutions will be mentioned and their different approaches and results will be observed. Hopefully this may act as a simple introduction to the area as such and a minor review of the current state of the art.

2 Background

This section will dive into the relevant areas of SCA and automatic program repair. The branch of runtime repair will not be covered. Thus, the main focus will be on what can be fixed "before" the execution of the code. However, it is worth mentioning there is lot of work being done in that field as well. Such as research on fault handling and system resilience with areas such as Chaos Engineering.

2.1 Static Code Analysis

According to the IEEE standard glossary static (code) analysis is the process of analysing the code without executing it [9]. Manual static code analysis can be done by developers, peer reviewing each others code. Another day-to-day static code analysis are the lints, the correction tools, available in most IDEs, Integrated Development Environments, nowadays. There are SCA tools with varying complexity and capabilities. Some implement basic rules on code-style and syntactic issues, while some are focused on security vulnerabilities, or common code smells and bugs [1] [2]. SCA tools, or as they are more commonly called SATs, are compared to automated program repair adopted in the industry and have much more research behind them. There is also a clear trend which shows that the common use is only increasing. With more and more adoption of the DevOps way of working, SATs have an indisputable integrated place in the workflow [8][10].

2.2 Automated Program Repair

In march 2019 Marcilio et al. published an article together with a dataset showing how the popularity of ASATs, automatic static analysis tools, is increasing, while the actual fixing of the issues discovered is left out. They present an disappointing number of 13% of the issues being resolved. Overall there was also an median of 18.99 days until the issues actually were solved. Which they claim is an improvement compared with previous studies [11].

As mentioned in the introduction there are multiple reasons for these numbers, and one possible approach for resolving more issues can be the use of automated program repairing tools.

There are a couple of different repair techniques used in practice today. The process demands a search over the space of changes done to the input space, the source code. Goues et al. generalizes these techniques into 3 broad categories:

- Heuristic Repair
- Constraint-based Repair
- Learning-based Repair

We can observe the different categories in figure 1 and the following subsections will briefly explain their different approaches to program patching.

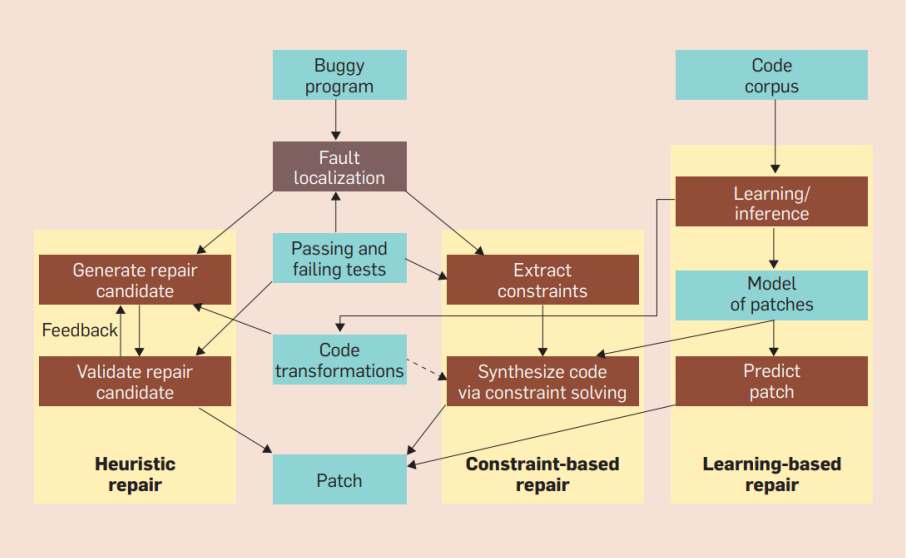

Figure 1: Overview of repair techniques. [7]

2.2.1 Heuristic Repair

The heuristic repair technique, as can be seen to the left in figure 1, uses a so called generate-and-validate approach. It consists of an iterative process where a search space of possible patches is constructed and then validated towards a set of tests. The patches are constructed by transforming the abstract syntax tree (AST). The AST is an representation of the source code as a tree, which mainly tries to identify and highlight the core parts of the code, while removing any unnecessary parts. There are three main components in the process of generating the patch code given the location of the possibly faulty code: mutation selection, test execution and traversal strategy.

Mutation selection is a combinatorial nightmare since the space of possible mutation could theoretically be infinite. Thus, the algorithm has to limit the possible edits to the patch to some extent.

In the *test execution* step patches are ran against the provided test cases to evaluate their success rate. It is quite a time-consuming exercise and thus selection and prioritization of test suites have been suggested as optimizations.

At last, different *traversal strategies* are applied on the search space. Meaning, in which direction, what order and what patches to explore[7].

2.2.2 Constraint-based repair

The Constraint-based repair technique uses a different approach where the desired constraints of the solution is generated and there after possible solutions are generated, which we can see in the center of figure 1. The idea is that the generated patch should be synthesized, and the optimal solution should be found by solving the generated constraints in the first step. This, can then be tested and different constraint suites can be generated based on the results of the patches generated to optimize the result on the tests [7].

2.2.3 Learning-based repair

Naturally, machine learning and deep learning techniques have found their way into this field as well. These methods can be used in symbiosis with the other two approaches and can be broken down into three categories.

The first one being learning from a corpus of code, containing "correct" code, which generates a likelihood of certain code snippets belonging to a bigger code corpus. It can then based on this model, rank different patches based on the likelihood they would occur in an existing "correct" code corpus.

The second approach generates templates of transformations based on previous commit history. It matches ASTs-to-ASTs from multiple which successfully passed previous static code analysis scans and can so generate a model for possible patching templates.

Finally there is one approach which covers the whole spectra end-to-end and doesn't rely on any specific information. There are a couple of different approaches. The idea is that a compiler is used as a judge of the output code from the network and the network just spits out different solutions. Then of

course this can and perhaps should be constrained in some manner, but could also constrain the network from finding some patch that is completely different than the more controlled algorithms present.

3 State of the Art

4 Discussion

There are some self-evident questions worth some reflections regarding the field of automatic program repair:

- Which are the known issues currently boggling the minds of the engineers developing these algorithms?
- As with the SCA, will people actually pick these tools up?
- How do we integrate them in a way such that they are used?
- What does the future have in store for us?

Firstly, there are some common issues discussed regarding the techniques used today for program repair. One of which is the issue of over-fitting the solutions...

References

[1] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at google. *Commun. ACM*, 61(4):58–66, March 2018. ISSN 0001-0782. doi: 10.1145/3188720. URL `https://doi.org/10.1145/3188720`.

[2] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 672–681. IEEE Press, 2013. ISBN 9781467330763.

[3] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[4] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018. ISSN 0360-0300. doi: 10.1145/3105906. URL `https://doi.org/10.1145/3105906`.

[5] Mark Harman Yue Jia, Ke Mao. Finding and fixing software bugs automatically with sapfix and sapienz. `https://engineering.fb.com/developer-tools/finding-and-fixing-software-bugs-automatically-with-sapfix-and-sapienz/` [Accessed: Apr 30 2019], Sep 2018.

[6] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. Getafix: Learning to fix bugs automatically. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360585. URL `https://doi.org/10.1145/3360585`.

[7] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, November 2019. ISSN 0001-0782. doi: 10.1145/3318162. URL `https://doi.org/10.1145/3318162`.

[8] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto. Automatically generating fix suggestions in response to static code analysis warnings. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 34–44, 2019.

[9] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[10] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald Gall, and Andy Zaidman. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 11 2019. doi: 10.1007/s10664-019-09750-5.

[11] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Canedo, Welder Luz, and Gustavo Pinto. Are static analysis violations really

fixed? a closer look at realistic usage of sonarqube. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 209–219. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00040. URL https://doi.org/10.1109/ICPC.2019.00040.