Improving Patch Quality by Enhancing Key Components of Automatic Program Repair

Mauricio Soto
Carnegie Mellon University
Pittsburgh, PA
mauriciosoto@cmu.edu

***Abstract*—The error repair process in software systems is, historically, a resource-consuming task that relies heavily in developer manual effort. Automatic program repair approaches enable the repair of software with minimum human interaction, therefore, mitigating the burden from developers. However, a problem automatically generated patches commonly suffer is generating low-quality patches (which overfit to one program specification, thus not generalizing to an independent oracle evaluation). This work proposes a set of mechanisms to increase the quality of plausible patches including an analysis of test suite behavior and their key characteristics for automatic program repair, analyzing developer behavior to inform the mutation operator selection distribution, and a study of patch diversity as a means to create consolidated higher quality fixes.**

***Index Terms*—Automatic Program Repair, Patch Quality**

I. Introduction

Software is pervasive and bugs in programs may have a significant impact in prominent areas of society. The cost of debugging software globally has risen to 312 billion dollars annually, and developers spend, on average, half of their time finding and repairing bugs [7]. Errors as such can compromise systems' security and privacy (e.g., Heartbleed [3]), and even cause death (e.g., Therac-25 medical radiation device [10]). Therefore, significant research efforts have focused on creating automatic program repair (APR) approaches [9], [12], [19] able to repair errors with minimum human interaction.

One well-known family of approaches, known as *generate-and-validate* repair, takes as input a program with a set of bugs and a test suite with passing and failing test cases describing correct functionality and desired but currently incorrect behavior (a bug). These approaches then *generate* variants of the original source code and *validate* them through the guiding test suite until a *plausible* patch is found (a variant that suffices the specification of the guiding test suite).

A common problem automatic program repair approaches face is the possible creation of low-quality plausible patches. This phenomenon occurs when the approach finds a variant that suffices the specification of the guiding test suite but it does not *generalize* to an oracle evaluation (e.g., a knowledgeable developer or a held-out independently created test suite). Our proposed work pursues to improve the generated patches' quality therefore closing the gap between theory and the application of these approaches in real-world systems.

Related Work: There have been previous efforts to increase the quality of candidate patches in automatic program repair improving upon previous approaches, however, not seeking to increase patch diversity nor increase the quality of plausible patches. HDRepair [11] modifies the fitness function based in fix history to assess patch suitability. The fitness of patch candidates is determined by how closely the changes in a patch occur in the analyzed corpus using a graph-based representation of the patches. Prophet [13] ranks candidate patches based on a probabilistic model mined from eight projects. GenProg [12], PAR [9] and TRPAutoRepair [19] are examples among a family of syntactic-based automatic program repair approaches seeking to generate patch candidates by modifying the program's syntax while semantic-based approaches use code synthesis to construct fix code. Test suite behavior in the context of automatic program repair has been studied in the C language with a corpus of programs written by novices [15].

There have been previous attempts to improve the quality of software by incentivizing diversity. N-Version Software is a way to take advantage of different implementations of code created following the same specification [2]. Smith et al. [15] compare the performance of single patches to hypothetical n-version patches, where the behavior of the n-version patches is described by a voting system where the n-version patch's behavior is determined by the behavior of the majority of the single patches. The work proposed in this paper leverages these previous ideas to create real n-version patches with actual compilable code to be executed by the test cases.

I, consequently, propose a set of mechanisms to improve the quality of plausible patches and thus the likelihood of generalizing to an oracle evaluation. My solution includes an analysis on the role of test suites in the repair process and how modifying characteristics of the guiding test suite leads to better quality patches, analyzing human behavior to inform the distribution of program edits and statement kinds as selected by APR approaches, and a study on patch diversity and how patch consolidation can be used as a tool to increase the quality of plausible patches.

Hence, my thesis statement is the following:

> Improving key components of the automatic program repair process such as test suite quality, mutation operator selection, and patch consolidation leads to an improvement in the quality of the produced plausible patches.

Some of the major contributions of this work are:

- Analysis of test suite characteristics and evaluation of APR approaches in real-world defects
- Creation of developer-informed repair approach
- Study of software diversity in the context of APR and creation of a multi-objective repair approach

The rest of this paper discusses three techniques to improve patch quality in APR, it proceeds as follows: Research Thrust I analyzes the role of test suites in automatic program repair, Research Thrust II depicts an study of developer patching behavior, Research Thrust III argues the potential of patch diversity and patch consolidation as a means to increase patch quality.

II. Research Thrust I: Analyzing the Role of Test Suites in the APR Process

In my first research thrust I hypothesize that enhancing key characteristics of the guiding test suite can lead to an improvement of the produced patches. More concretely, we conducted an experiment where we modify the coverage, size, and provenance, among other characteristics from the guiding test suites with the intent of analyzing which quality features from the test suite have higher impact in patch quality.

We generated a set of plausible patches from the corpus Defects4J [8] since it is a database and extensible framework of real-world bugs that enables reproducible studies in software testing and has been previously used to evaluate automatic program repair [14], [18].

To create plausible patches, we created a publicly available Java Repair Framework (JaRFly)[1] which includes three APR approaches for Java: GenProg [12], PAR [9] and TRPAutoRepair [19]. Since generate-and-validate approaches rely in stochastic processes, we executed each bug in the corpus using 20 different seeds with a 4 hour budget per each seed. We found a total of 1,298 plausible patches from 68 bugs.

To measure the relationship between test suite coverage and repair quality, where $coverage$ is measured using statement coverage in the defective version of code by the training test suite, and $size$ is measured as the number of test cases in the training test suite, we created subsets of test cases from the developer-written test suite of varying coverage.

One principled way to evaluate patch quality in automatic program repair is by using held-out test suites as a quality mechanism checking if the plausible patches generalize to a different instance of the same specification [15]. To analyze the quality of the plausible patches generated we created an independent automatically generated held-out test suite from the developer's patch using Evosuite [6], and executed these held-out test suites on the patches generated using the subsets of test cases of varying coverage.

For GenProg patches, the training test suite coverage was significant at $p < 0.1$ but not significant at $p < 0.05$; while the training test suite size was significant at $p < 0.05$. For PAR, both, training test suite coverage and size were significant at

[1] https://github.com/squaresLab/genprog4java

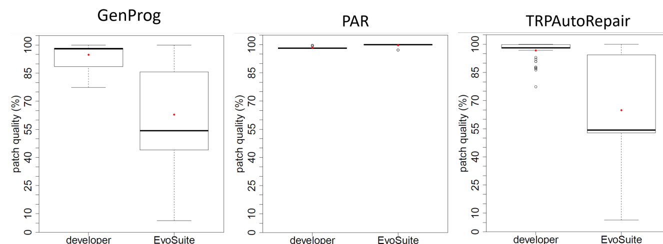

Fig. 1. Using EvoSuite-generated test suites for program repair resulted in fewer patches than those generated using the developer-written test suites. The box-and-whisker plots compare the quality of the defect populations. The horizontal line represents the median and the red dot shows the mean.

$p < 0.05$. For TRPAutoRepair, the training test suite coverage was significant at $p < 0.1$ but not significant at $p < 0.05$; while the training test suite size was significant at $p < 0.05$. These results provide evidence that there is significant effect of the training test suite size on the quality of the patches produced using automatic program repair techniques and a less significant effect of the training test suite coverage.

To measure the association between test suite provenance and patch quality, we executed the repair techniques using the EvoSuite-generated tests and measure their quality using the independent, developer-written test suite. To control for defects, we consider only program versions that can be patched using test suites from both provenances.

Figure 1 shows the quality of the patches generated using the two provenances. The box-and-whisker plots show the distribution of patch quality. The Mann-Whitney U test rejects the null hypothesis that states that these populations do not differ ($p = 2.98 \times 10^{-6}$ for GenProg, $p = 1.51 \times 10^{-4}$ for PAR, and $p = 6.2 \times 10^{-7}$ for TrpAutoRepair). The *delta estimate* computed using Cliff's Delta test shows that median patch quality of the patches produced using EvoSuite-generated test suites is lower for GenProg and TrpAutoRepair whereas it is higher for PAR. The 95% confidence interval (CI) does not span 0 for all three techniques indicating that, with 95% probability, two populations are likely to have different distributions.

These results show that the provenance of the guiding test suite has a significant effect on repair quality. Our conclusion is consistent with earlier findings [15]. However, our results indicate that the effect may not be the same for all techniques.

III. Research Thrust II: Analyzing Developer Behavior to Inform Automatic Program Repair

In my second research thrust, I hypothesize that mimicking human behavior selection decisions instead of using the current heuristic-based approach would lead to higher quality patches. These heuristics are mostly based in general approximations of reasonable behavior and have not been carefully calibrated. My intuition is that since developers have a wide understanding of what edits and statements need to be selected to fix errors, analyzing developer behavior to fine-tune

the selection decisions in APR would increase the produced patches' quality.

In this study we mined a substantial corpus of bug fixing commits created by human developers from popular projects in GitHub. We then analyzed the distribution of mutation operators as used per developers by comparing the differences between the AST representation of the before-fix version and the after-fix version, and matching these differences to the mutation operators used by state-of-the-art APR approaches. We then designed and executed an experiment to compare APR approaches using this distribution against approaches using the heuristic-based distribution showing that APR tools informed by human behavior find higher quality patches faster in most analyzed cases. This work has been completed and published [18].

In other of our related studies, I have analyzed prominent corpora of bug fixing commits creating association rules to determine what statement types [17] and mutation operators [18] are used commonly together by developers to inform multi-edit patches and how there is a tension between the quantity and the confidence of the generated rules to be used in APR. We also analyze how developers replace statement kinds, and with what frequency each statement kind is replaced, deleted or added [16]. We conclude that a mutation operator selection mechanism informed by developer behavior creates higher quality patches faster than its heuristic-based counterpart and that automatic program repair benefits from having a diverse mutation operator pool.

IV. Research Thrust III: Patch Diversity and Consolidation

In my third research thrust I hypothesize that consolidating plausible patches leveraging semantic and syntactic diversity might lead to higher quality fixes. The intuition behind this idea is that each plausible patch is a partial solution that fails to cover all possible cases from the specification, therefore consolidating several plausible patches might increase the number of cases described by the specification and thus the quality of the overall consolidated solution. I have conducted an initial experiment showing the success of patch consolidation in a subset of the corpus described in the previous section by consolidating the generated plausible patches using the off-the-shelf software merging tool JDime [1].[2]

First, we create all possible combinations of two distinct patches using off-the-shelf combination mechanisms provided by Jdime: Line-based (similar to GitHub merge), and Structured (a merge considering the AST).

We then used held-out test suites to evaluate the quality of the consolidated patches product of all the combinations of two patches per each bug. Figure 2 describes the behavior of the consolidated patches. Bars describe the quality of the combinations that perform better than the individual patches using the left axis, and the line describes the number of combinations created per each approach using the right axis.

[2]https://github.com/se-passau/jdime

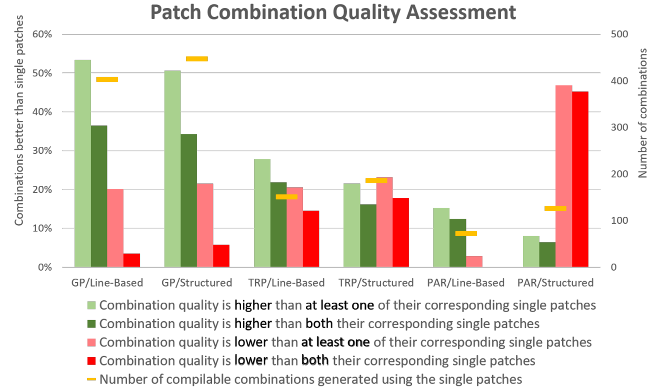

Fig. 2. Quality assessment of patch combinations and their corresponding individual plausible patches

The results show that up to 54% of the consolidated patches show higher quality than at least one of their corresponding individual plausible patches, and up to 37% of the consolidated patches show higher quality than both of their corresponding individual plausible patches. This shows that there is a considerable potential in patch consolidation as a means to improve plausible patch quality. Next, I propose the future work:

A. *Improve Diversity of Generated Patches*

One problem I have found in the population of plausible patches we have generated is the lack of diversity in the sample. It is common to find patches that are semantically identical based on manual inspection. This plausible patch generating behavior shown in automatic program repair tools could be improved by incentivizing the search for diversity. I propose to enhance the methodology to traverse the search space of patch candidates in an effort to increase the semantic and syntactic difference of plausible patches, therefore amplifying the potential of increasing quality by patch consolidation.

1) Modify the Fitness Function to Incentivize Diversity: Several current approaches use genetic programming to find plausible patch candidates. Genetic programming relies on a fitness function used to compute the likelihood of patch candidates to be plausible patches. Current approaches set their fitness function to look mainly for patch correctness by assigning a fitness score based on the number of passing test cases. This fitness score determines the candidate's likelihood to be selected in future generations.

One way to incentivize diversity when traversing the search space is by modifying the fitness function to look not only for patch correctness, but also for semantic diversity between the candidate patches. I propose a fitness function based in multi-objective search to look for both correctness and diversity therefore selecting the Pareto-optimal patch candidates which excel in both features as opposed to the the current one-dimensional approach.

To identify patch diversity there needs to be a quantitative measurement to analyze *how different* is a program with

respect to another. Since program equivalence is an undecidable problem, I propose four quantitative measurements which approximate program equivalence and diversity.

Test-suite based semantic difference: Figure 3 shows a diagram of the proposed process to measure semantic difference. It starts with two programs: Program A and Program B. The goal is to determine how semantically different are Program A and Program B. Step 1: create a specification from each of the programs. In this case the specifications take the form of a test suite generated based on the program. This may be achieved using test suite generation tools (e.g., Evosuite [6], Randoop).

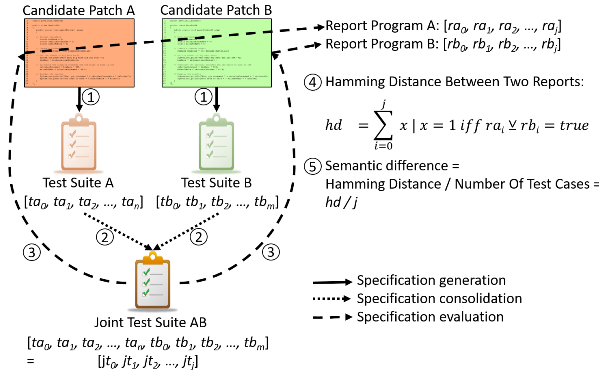

Fig. 3. Semantic difference measurement

Step 2, both specifications are consolidated and this consolidated specification is used to evaluate each individual program (Step 3). The evaluations are then compared using Hamming distance, and the final score is computed as the number of test cases that behave differently in both candidate patches as a percentage of all test cases.

2) Slicing Mutation Operators and Fault Space: Another possible way to increase the diversity in plausible patches is by restricting the search space to different clusters therefore trying to find local optima in different sections of the search space instead of seeking the global optimum every time. I propose to restrict the search space in three different ways: Slicing mutation operators, fault locations, and test cases.

APR approaches use edits, usually referred to as mutation operators, to create patch candidates. When slicing mutation operators I will restrict our APR approaches to only use non-overlapping sets of mutation operators. This will force the approach to look for a patch candidate that uses the mutation operators in a given set only. Similarly, when slicing fault locations and test cases, the approaches will be forced to look for candidate patches considering only a set of given fault locations or test cases therefore increasing the diversity of solutions.

B. Timeline

The proposed timeline shown in Figure 4 includes the two most relevant previous projects and proposed following tasks.

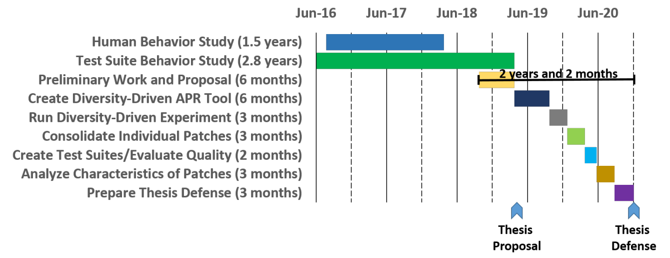

Fig. 4. Proposed timeline for thesis

V. Conclusion

Automatic program repair is a promising area with significant results. Nevertheless, a common problem these approaches suffer is the creation of low-quality plausible patches (which overfit to the guiding test suite). We, therefore, propose a set of mechanisms to improve the quality of plausible patches. The proposed future work focuses in increasing patch diversity generated in APR and improve the quality of said patches through patch consolidation. An initial experiment shows that patch consolidation leads to an increase in quality over one single patch in 54% of the consolidated patches, and an increase in quality over both individual patches in 37% of the consolidated patches.

References

[1] S. Apel, O. Leenich, and C. Lengauer. Structured merge with auto-tuning: balancing precision and performance. In ICSE 2012.
[2] A.A. Avizienis. The Methodology of N-version Programming. Software Fault Tolerance, edited by M. Lyu, John Wiley & Sons, 1995.
[3] M. Carvalho, J. DeMott, R. Ford, D. A. Wheeler. Heartbleed 101. In IEEE Security Privacy. Volume 12, Issue 4, 2014
[4] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. Quickly detecting relevant program invariants. In ICSE 2000, pp. 449–458.
[5] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In ASE 2014.
[6] G. Fraser and A. Arcuri. Evosuite: automatic test suite generation for object-oriented software. In ESEC/FSE 11, pages 416-419.
[7] P. Goodliffe. Becoming a Better Programmer. OReilly Media, Inc., 2014.
[8] R. Just, D. Jalali, and M. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In ISSTA 2014.
[9] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In ICSE 2013, pages 802-811.
[10] N. G. Leveson, C. S. Turner. An Investigation of the Therac-25 Accidents. In IEEE Computer,
[11] X.-B. D. Le, D. Lo, and C. Le Goues. History driven program repair. In SANER, 2016.
[12] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. In IEEE TSE 2012.
[13] F. Long and M. Rinard. Automatic patch generation by learning correct code. In POPL'16, pages 298-312, 2016.
[14] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. In Springer Empirical Software Engineering, 2016.
[15] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In ESEC/FSE15.
[16] M. Soto, F. Thung, C. Wong, C. Le Goues, and D. Lo. A Deeper Look into Bug Fixes:Patterns,Replacements,Deletions,and Additions.MSR'16.
[17] M. Soto and C. Le Goues. Common Statement Kind Changes To Inform Automatic Program Repair. In MSR 2018.
[18] M. Soto and C. Le Goues. Using a Probabilistic Model to Predict Bug Fixes. In Proceedings of SANER 2018.
[19] Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In ICSM 2013, pages 180-189.