Ph.D. Dissertation Proposal

Improving Patch Quality by Enhancing Key Components of Automatic Program Repair

Mauricio Soto
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA
mauriciosoto@cmu.edu

April, 2019

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Thesis Committee

Claire Le Goues (Chair) Christian Kaestner
Institute for Software Research Institute for Software Research
Carnegie Mellon University Carnegie Mellon University

Nicholas A. Kraft William Klieber
Lead Principal Scientist Software Security Researcher
ABB Corporate Research Software Engineering Institute

Copyright © 2019 Mauricio Soto
Contents

1 Introduction 2

2 Thesis 4
  2.1 Contributions ................................................................. 5
  2.2 Success Criteria ............................................................ 6
  2.3 Potential Applications .................................................... 7

3 Research Thrust I: Analyzing the Role of Test Suites in the APR Process 7
  3.1 Generate Plausible Patches from a Substantial Corpus of Bugs ........... 8
  3.2 Guiding Test Suite Coverage and Size ................................... 9
  3.3 Guiding Test Suite Provenance .......................................... 11
  3.4 Number of Failing Tests in the Guiding Test Suite ...................... 13

4 Research Thrust II: Analyzing Developer Behavior to Inform APR Selection Mechanisms 15
  4.1 Analysis of Developer Behavior in Java Projects ....................... 15
  4.2 Corpus Mining from Popular Github Projects .......................... 16
  4.3 APR Tool Informed by Human Behavior ................................ 17

5 Research Thrust III: Patch Diversity and Consolidation 18
  5.1 Preliminary Work ........................................................... 20
  5.2 Improve Diversity of Generated Patches ................................ 21
    5.2.1 Modify the Fitness Function and Implement Multi-Objective Search to Incentivize Diversity ...................................................... 23
    5.2.2 Proposed Test-Suite-Based Semantic Difference Measurement .......... 23
    5.2.3 Slicing Mutation Operators, Fault Space, or Test Cases ........... 24
  5.3 Analyzing Meaningful Characteristics of Consolidated Patches ........... 25
    5.3.1 Obtaining Patch Quality Corpus ................................... 25
    5.3.2 AST-Based Syntactic Difference ..................................... 25

6 Research Plan 26
  6.1 Tasks to Completion ....................................................... 26
  6.2 Timeline ............................................................................. 27
  6.3 Risks .................................................................................. 27

7 Related Work 29
  7.1 Search-Based Approaches .................................................. 29
  7.2 Semantic-Based Techniques ................................................. 30
  7.3 Software Diversity ............................................................ 31

8 Conclusion 32
Abstract

The error repair process in software systems is, historically, a resource-consuming task that relies heavily on manual developer effort. Automatic program repair approaches enable the repair of software with minimum human interaction mitigating the burden on developers. However, a problem automatic program repair approaches commonly suffer is generating low-quality patches which overfit to one program specification as described by the guiding test suite, thus not generalizing to an independent oracle evaluation. This work proposes a set of mechanisms to increase the quality of plausible patches (which pass all the test cases in the guiding test suite but do not generalize to an independent evaluation), including an analysis of test suite behavior and their key characteristics for automatic program repair (Research Thrust I), analyzing developer behavior to inform the mutation operator selection distribution (Research Thrust II), and a repair technique based on patch diversity as a means to create consolidated higher quality fixes (Research Thrust III).

1 Introduction

Bugs in programs can have a significant impact in prominent areas of society given the pervasive nature of software systems. The cost of debugging software globally has risen to $312 billion annually, and developers spend, on average, half of their time finding and repairing bugs [23]. Errors as such can compromise systems’ security and privacy (e.g., Heartbleed [10]), and even cause death (e.g., Therac-25 medical radiation device [40]).

Repairing bugs like these is one of the most resource-intensive tasks in software development, requiring substantial manual effort [76, 71]. Therefore, significant attempts have been dedicated in the last several years to create automatic program repair (APR) approaches, which are able to repair errors with minimum human interaction [39, 31, 75, 41, 42, 16, 60, 74]. One well-known family of approaches known as generate-and-validate is described in Figure 1. This proposal will focus on this group of APR techniques. Other types of repair approaches exist in the literature, such as semantic-based repair where tools use code synthesis to construct code fixes based on constraints.

Generate-and-validate repair takes as input a program with one or more bugs, and a test suite with passing and failing test cases (Phase 1 in Figure 1). The passing test cases describe correct program functionality that should be maintained, while failing test cases specify incorrect program behavior. All test cases in the test suite are assumed to be correct.

Generate-and-validate approaches then identify the locations of the program with higher probability of being buggy (Phase 2) by applying mechanisms from fault localization literature (e.g., spectrum-based fault localization techniques such as Tarantula [26]). These approaches then generate variants of the original source code (Phase 3), usually referred to as patch candidates, based on their available mutation operators. There is a broad diversity of such operators used in automatic program repair, including deleting or inserting statements [39, 62], applying templates [30], transformation schemas [41, 42], or semantically-inferred code [14, 49, 48].

Finally, the approaches validate the patch candidates by executing the test suite (Phase 4). If a variant is found that satisfies the behavior described by the test suite, this variant is
considered a plausible patch (Step 5), where “plausible” indicates that the variant passes all test cases. These patches have therefore a higher probability of being a correct patch for the bug [64]. These approaches have been successful in patching bugs for real-world software systems [45 39 30], as well as simpler software created for educational purposes [68]. Researchers have proposed several instances of this family of approaches as successful exponents of APR achievements [39 30 75 62].

Figure 1: The automatic program repair generate-and-validate family to produce plausible patches starting from source code containing a bug and a test suite that describes the desired program behavior with passing and failing test cases.

The possible creation of low-quality plausible patches is a fundamental problem automatic program repair approaches face [64]. This phenomenon occurs when the approach finds a variant that satisfies the partial specification provided by the guiding test suite, but it fails to generalize to an independent evaluation (e.g., a knowledgeable developer or a held-out test suite). Guiding test suites are intrinsically incomplete, since they describe discrete samples in the behavior space, such that plausible patches can satisfy all provided tests but fail to satisfy an independent evaluation. Our proposed work aims to improve the generated patches’ quality, making these approaches a more powerful tool usable in real-world systems.

Concretely, the generate-and-validate process is a versatile set of steps that several approaches have instantiated [39 30 75 62], and some of its components can be enhanced to increase the probability of the plausible patches generated by these tools to be correct patches.

The rest of this document proposes 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 and how modifying characteristics lead to better quality patches; Research Thrust II depicts an study of developer behavior to inform the distribution of program edits and statement kinds as selected by APR approaches, and Research Thrust III argues the potential of patch diversity and patch consolidation as a means to increase patch quality.
2 Thesis

My research seeks to improve patch quality in automatic program repair by improving key components of the generate-and-validate process. I have identified three segments in the APR methodology that can benefit from specialized improvements leading to higher quality patches, recognized as Phases 1, 3, and 5 in Figure 1. Previous studies [68, 70, 42] show the potential of these components to maximize impact in the quality of the generated plausible patches. The remaining sections of the APR process have either already been analyzed in detail in the past (e.g., fault localization techniques [26, 2, 58, 1, 33] in Phase 2), or have focused in improving the speed of finding patches (e.g., test prioritization [75] in Phase 4).

• Guiding test suite:
  Generate-and-validate repair relies on a partial specification of the desired fixed program, commonly taking the form of a test suite (a set of test cases describing the expected program behavior; we usually refer to it as the “guiding” test suite). Our work analyzes fundamental test suite characteristics such as test suite coverage, provenance, and number of triggering test cases to determine which of these have a higher impact in the quality of the resulting plausible patches. We selected a corpus of real-world bugs and modified different characteristics of the guiding test suites in this corpus, and create patches using off-the-shelf APR techniques. We then evaluate the generated patches to determine which characteristics most influence patch quality.

• Mutation operator selection:
  State-of-the-art generate-and-validate APR techniques select between and instantiate various mutation operators to construct candidate patches, informed largely by heuristic probability distributions, which may reduce effectiveness in terms of both efficiency and output quality due to the inaccurate nature of heuristics. In practice, human developers use some mutation operators far more often than others. We, therefore, propose an approach to guide the mutation operator selection mechanism in automatic program repair by analyzing and mimicking human developer behavior thus improving the quality of generated patches.

• Diversity-driven repair and plausible patch consolidation:
  Several generate-and-validate approaches rely on stochastic processes [39, 75, 62], therefore, it is common to obtaining several plausible patches for a single defect. I propose a repair approach that uses multi-objective search to optimize for diversity and correctness to increase the diversity of patches found. A diverse pool of implementations reduces the probability of identical fault behavior therefore providing higher fault tolerance. Finally, I propose patch consolidation as a mechanism to merge diverse plausible patches leading to an increase in the quality of the generated plausible patches.

Hence, the following thesis statement summarizes the proposed principle claim of this research:
Thesis statement: Automatic program repair approaches may create low quality plausible patches that overfit to the guiding test suite. Improving key components of the automatic program repair process (specifically, test suite quality, mutation operator selection, and a patch consolidation repair technique that is able to exploit the semantic diversity in automatically generated patches) leads to an improvement in the quality of the produced patches.

2.1 Contributions

The major contributions of this proposed dissertation are:

- **Analysis of test suite characteristics:** An analysis of the role test suites play in the context of automatic program repair. We analyze fundamental characteristics of test suites and the impact these characteristics have in the obtained patches’ quality measures.

- **Evaluation of APR approaches in real-world defects:** We evaluate three state-of-the-art APR approaches in a set of 357 real-world defects from open-source projects which provides empirical information about the usage of APR in a real-world environment.

- **Analysis of patch-fixing developer behavior:** This thesis proposal provides a deeper understanding of human developer behavior when fixing errors in source code and frequency of the analyzed mutation operators.

- **Creation of developer-informed repair approach:** We mined an empirical model of single-edit repairs from a substantial corpus of open-source projects. We later used this knowledge to inform an APR approach favoring mutation operators which human developers more commonly use.

- **Formulation of specification-based measure to approximate program equality and diversity:** We propose a measurement based in program specifications to approximate program equality and thus, its trivial opposite, program diversity.

- **Study of software diversity in the context of APR:** The third thrust of this proposal focuses in program diversity optimization in the context of APR. This is beneficial, both, to find more diverse patches thus avoiding similar faulty behavior, and to exploit these diverse patches to consolidate solutions into a higher quality fix.

- **Creation of multi-objective repair approach:** A proposed multi-objective repair approach which incentivizes the search space traversal to optimize for correctness and diversity, therefore finding a more diverse pool of patches to be consolidated thus improving patch quality.
• **Development and integration**: We develop these approaches for the Java programming language and integrate the repair approaches into a publicly available APR tool. All the code and data produced in this thesis proposal to run our experiments is made publicly available to support reproducibility and extension.

### 2.2 Success Criteria

The main goal of this thesis proposal is to create higher quality patches by enhancing key components of the automatic program repair process. In this context, **patch quality** becomes a fundamental concept that must be measurable and quantifiable.

Since software system functionality is described by subjective human requirements, determining whether one patch is “better” than another is often difficult to assess. A perfect oracle would be able to check the program formally against a full specification, but given the nonexistence of such specifications and oracles in practice, we are forced to find alternatives. We can approximate this oracle by creating an specification (e.g., independent evaluation test suite [68]) based on the fixed version of code generated by the developer working on the analyzed project when the selected error was patched. The concept of comparing test behavior in two versions of code has been previous used in differential testing [47] and fault localization [13, 79]. Figure 2 shows the process to create the evaluation held-out test suite and use it to assess the quality of plausible patches generated by APR approaches.

![Diagram of the process](image)

**Figure 2**: Evaluating the quality of generated plausible patches based on a held-out test suite generated from a developer patch.

Figure 2 starts with a buggy program that is later patched by a knowledgeable developer (who understand the expected functionality of the program). We then use off-the-shelf test suite creation tools (e.g., Evosuite [20], Randoop [55]) to generate a held-out test suite that describes the behavior of the oracle patch. Independently, we use automatic program repair techniques to generate plausible patches for these bugs. Finally we use the held-out test suite to evaluate the plausible patches. Since our quality metric is based in semantic similarity to the developer patch, the percentage of test cases from the plausible
patch that pass this evaluation is proportional to the quality of the patch under evaluation (higher is better). Previous studies [68, 69, 34] have used this quality metric to evaluate APR patches, and we will use it throughout the quality evaluation of patches generated in this proposal. This methodology provides a more objective assessment of functional patch quality than a single human rater can and removes possible subconscious bias [36].

2.3 Potential Applications

The research proposed in this dissertation has applications that extend to real-world industrial software systems using APR techniques. The popularity of APR in the research environment continues to grow, and similarly, applications in industrial environments using APR have started to emerge [45]. However, current approaches [39, 78, 69, 30, 62] still create a considerable number of low-quality patches which undermines the adoption of APR tools. This study proposes a set of techniques to increase the quality of plausible patches found in the APR process, therefore diminishing the gap existing between state-of-the-art APR approaches and their broader adoption in real-world applications.

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

A fundamental component in the automatic program repair process is the guiding test suite. This test suite works as a partial specification of the desired program showing both correct behavior to keep (positive test cases), and erroneous behavior (negative test cases). The triggering criteria to declare a variant to be a plausible patch is when all test cases in this test suite pass. A low quality guiding test suite might easily lead the APR approach to create plausible but incorrect patches (which generate correct outputs for all test cases but incorrect outputs for other inputs [42]).

In this research thrust, we 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 on patch quality. Future projects interested in using automatic program repair can optimize for these characteristics in their corresponding test suites.

In the following sections of this thrust we first generate a baseline of patches using a substantial corpus of bugs (Section 3.1), and consequently we modify the coverage and size (Section 3.2), provenance (Section 3.3), and number of failing tests (Section 3.4) of the guiding test suites from the analyzed bugs to reason about how the test suite characteristics relate to patch quality.
3.1 Generate Plausible Patches from a Substantial Corpus of Bugs

We generated a set of plausible patches for bugs from the corpus Defects4J [27]. Defects4J is a database and extensible framework of real-world bugs from popular real-world projects that enables reproducible studies in software testing and has been previously used to evaluate automatic program repair [46, 69]. Each bug in Defects4J fulfills the following requirements: The bug is explicitly labelled as a bug fixing commit and it is related to source code, the code includes at least one test case that shows the buggy behavior and it is reproducible, the bug is isolated (does not include code changes unrelated to the bug fix). The distribution of analyzed bugs from Defects4J is described in Table 1.

<table>
<thead>
<tr>
<th>Project</th>
<th>KLoC</th>
<th>Defects</th>
<th>Tests</th>
<th>Test KLoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure Compiler</td>
<td>85</td>
<td>133</td>
<td>3,353</td>
<td>75</td>
</tr>
<tr>
<td>Commons Lang</td>
<td>19</td>
<td>65</td>
<td>173</td>
<td>31</td>
</tr>
<tr>
<td>Commons Math</td>
<td>84</td>
<td>106</td>
<td>212</td>
<td>50</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>85</td>
<td>26</td>
<td>222</td>
<td>42</td>
</tr>
<tr>
<td>JodaTime</td>
<td>29</td>
<td>27</td>
<td>2,599</td>
<td>50</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>302</strong></td>
<td><strong>357</strong></td>
<td><strong>6,559</strong></td>
<td><strong>248</strong></td>
</tr>
</tbody>
</table>

Table 1: 357 defects from five real-world projects in the Defects4J version 1.1.0 benchmark. Project refers to the name of the real-world project under analysis. KLoC describes the size of each project in thousands of lines of code. Defects describes the number of bugs per project. The Tests describes the number of tests and Test KLoC column refers to the length of developer-written tests.

To create plausible patches, we used three well-known state-of-the-art APR approaches: GenProg [39], PAR [30], and TRPAutoRepair [62]. Since the implementation described in the original publication of PAR is not publicly available [30], we modified the publicly available automatic program repair tool GenProg4Java\(^1\) to include the mutation operators of PAR and search criteria of TRPAutoRepair [62] in addition to the functionality already included in the tool (GenProg [39]). Since these generate-and-validate approaches rely in stochastic processes, we executed each APR approach using each bug in the corpus using 20 different seeds with a 4 hour budget per each seed following guidelines from previous studies [39, 69]. We found a total of 1,298 plausible patches for 68 bugs as described in Figure 3.

We created an independent automatically generated held-out test suite from the developer’s patch to analyze the quality of the plausible patches generated using Evosuite, an automated unit test generation tool which creates tests optimized for code coverage applying search-based techniques [19, 20]. (Section 2).

54.9% of the plausible patches created for the GenProg bugs generalize to the held-out test suite. Figure 4 describes the results of the quality assessment performed to the individual plausible patches of the remaining approaches.

\(^1\)https://github.com/squaresLab/genprog4java
Figure 3: GenProg, PAR, and TRPAutoRepair produce a patch 1,298 times out of 21,420 attempts, and at least one technique can produce a patch for 68 out of the 357 defects.

<table>
<thead>
<tr>
<th>APR Technique</th>
<th>Total Patches</th>
<th>Unique Patches</th>
<th>Defects Patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenProg</td>
<td>586</td>
<td>255</td>
<td>49</td>
</tr>
<tr>
<td>PAR</td>
<td>199</td>
<td>107</td>
<td>38</td>
</tr>
<tr>
<td>TRPAutoRepair</td>
<td>513</td>
<td>199</td>
<td>44</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1,298</strong></td>
<td><strong>561</strong></td>
<td><strong>68</strong></td>
</tr>
</tbody>
</table>

Figure 4: The quality of the plausible patches generated using the developer-written test suite.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Patch Quality</th>
<th>100%-Quality Failed at least one test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Min</td>
<td>Mean</td>
</tr>
<tr>
<td>GenProg</td>
<td>64.8%</td>
<td>95.7%</td>
</tr>
<tr>
<td>PAR</td>
<td>64.8%</td>
<td>96.1%</td>
</tr>
<tr>
<td>TRPAuto Repair</td>
<td>64.8%</td>
<td>96.4%</td>
</tr>
</tbody>
</table>

We then manipulated the guiding test suites to vary coverage, size, provenance, and number of failing test cases to evaluate their effect on output patch quality as described in the following sections.

### 3.2 Guiding Test Suite Coverage and Size

How does the coverage of the test suite used to produce the patch affect patch quality? How does the size of the test suite affect patch quality?

**Creating test suite subsets varying coverage and size:** To measure the relationship between test suite coverage and repair quality, where coverage is measured as the percentage of lines covered 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. We then used the repair techniques to produce patches using these test suites, and then computed the quality of the provided patches (Figure 2).

To create the test suite subsets, we first compute the minimum and the maximum code coverage ratio of each developer-written test suite. The minimum code coverage ratio \((\text{cov}_{\text{min}})\) is the statement coverage on the defective code version of just the failing test cases, these describe the behavior to be corrected by APR. The maximum code coverage ratio \((\text{cov}_{\text{max}})\) is the statement coverage of the entire developer-written test suite on the defective code version. The potential test suite coverage variability is the difference between
the minimum and the maximum: $\Delta_{\text{cov}} = \text{cov}_{\text{max}} - \text{cov}_{\text{min}}$. Defects whose variability $\Delta_{\text{cov}}$ is less than 25% lacks sufficient variability to be used in this study. We thus discard 18 out of the 68 defects that had at least one repair technique produce at least one patch.

For each of the 50 remaining defects, we chose five target coverage ratios evenly spaced between the minimum and the maximum. For each target ratio $c$, we attempted to create five distinct test suite subsets within a 5% margin of $c$ using \textsc{SampleTestSuiteCoverage} (Algorithm 1). Each test suite subset starts with all the failing tests and iteratively attempts to add a uniform randomly selected passing test case without replacement, as long as it does not make the subset’s coverage exceed the target by more than 5%. The process succeeds if the subset reaches within 5% of the target coverage. If after attempting to add a test 500 times the target is not reached, the process restarts. We repeat this process until we achieve five \textit{distinct} test suites for each coverage target.

For five of the 50 defects the sampling algorithm was unable to generate five distinct test suite subsets, so we discard these five defects and use the remaining 45 defects for the experiments.

For each of the 45 defects, we had 25 test suite subsets (five for each coverage target), and we attempted each repair 20 times on different seeds. In total, these 22,500 repair attempts produced 11,750 patches. Figure 5 shows the patch distribution. GenProg produced at least one patch for 40 out of 45 defects, PAR 36, and TRPAutoRepair 40.

To evaluate quality we created held-out test suites using the mechanism in Section 2.2. Our goal is to have high quality held-out test suites, therefore we only evaluated the patches for which we were able to create a held-out test suite that shows 100% line coverage in the method modified in the oracle patch and at least 80% class coverage in the oracle modified class. Figure 6 shows, for each technique, the minimum, mean, and maximum quality achieved by this technique, and what percentage of the generated patches achieved full quality. Next, for each technique, we built a multiple linear regression model to predict the quality of the patches based on the training test suite coverage and size.

\textbf{Results}: The quality of patches produced by GenProg is equal to $94.78 + 0.02(\text{coverage}) + 10$
Figure 5: Distribution of the number of patches produced using developer-written training test suite subsets of varying code coverage on the defective code version.

<table>
<thead>
<tr>
<th>technique</th>
<th>minimum</th>
<th>mean</th>
<th>maximum</th>
<th>100%-quality patches</th>
<th>Failed at least one test</th>
</tr>
</thead>
<tbody>
<tr>
<td>GenProg</td>
<td>0.00%</td>
<td>94.79%</td>
<td>100.0%</td>
<td>19.34%</td>
<td>80.66%</td>
</tr>
<tr>
<td>PAR</td>
<td>51.82%</td>
<td>90.56%</td>
<td>100.0%</td>
<td>15.46%</td>
<td>84.54%</td>
</tr>
<tr>
<td>TRPAutoRepair</td>
<td>62.92%</td>
<td>95.51%</td>
<td>100.0%</td>
<td>21.62%</td>
<td>78.38%</td>
</tr>
</tbody>
</table>

Figure 6: Patch quality results: The quality of patches varied from 0.0% to 100.0%.

0.01(size). The training test suite coverage was significant on \( p < 0.1 \) but not significant on \( p < 0.05 \); while the training test suite size was significant on \( p < 0.05 \). The quality of patches produced by PAR is equal to 90.56 – 0.11(coverage) + 0.03(size). Both, training test suite coverage and size were significant on \( p < 0.05 \). The quality of patch produced by TRPAutoRepair is equal to 95.50 + 0.01(coverage) + 0.005(size). The training test suite coverage was significant on \( p < 0.1 \) but not significant on \( p < 0.05 \); while the training test suite size was significant on \( p < 0.05 \).

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. Having large number of relevant tests can increase the quality of patches. The results obtained for training test suite coverage do not allow us to make any concrete claims about the effect of training test suite coverage on the quality of the patches. This is consistent with previous findings [68, 53].

### 3.3 Guiding Test Suite Provenance

How does the provenance of the test suite (developer-written vs. automatically generated) influence patch quality?

Creating test suites from different provenances: 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 both test suites.

For repair techniques to be able to repair a defect, the test suite they use must evidence that defect. We analyzed the EvoSuite-generated test suites for the 68 defects for which at least one technique produced at least one patch, to identify which of these generated test suites contained at least one test that fails on the defective code version of the program. We discarded the 31 defects that had no such failing test cases. The remaining 37 defects have at least one generated failing test.

We executed each of the three repair techniques on each of the 37 defects resulting in 740 repair attempts per technique. We identified the sets of defects where both test suites (developer-created and automatically-generated) led a technique to find a repair. We call these the in-common populations.

Figure 7: 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 in-common defect populations. The horizontal line represents the median and the red dot shows the mean. The quality of patches produced by GenProg and TRPAutoRepair using the EvoSuite-generated test suites is statistically significantly lower that those produced using developer-written ones. The quality of patches produced by PAR using the EvoSuite-generated test suites is statistically significantly higher that those produced using developer-written ones. These results are statistically significant using Mann-Whitneys U test, confirmed using confidence intervals and Cliffs Delta.
Results: Figure 7 summarizes these results. Using EvoSuite-generated test suites for program repair results in far fewer patches generated. Between 3.38% (PAR) and 16.35% (GenProg) of the attempts resulted in a patch, whereas using developer-written test suites resulted in between 10.27% (PAR) and 21.49% (GenProg) of the repair attempts producing patches. Similarly, far fewer defects were patched when using EvoSuite-generated test suites (5.41%–40.54% vs. 54.05%–81.08%).

We observe that the minimum, mean (not shown), and median quality of the patches generated using automatically generated test suites for GenProg and TrpAutoRepair are lower than that of the patches produced using the developer-written test suites (the maxima were always the same). For PAR, we observe that mean, median, and maximum patch quality is higher while the minimum patch quality is lower for the patches produced using the automatically generated test suites than those produced using developer-written ones.

Figure 7 also shows the quality of the patches generated using the two provenances for the in-common populations. 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 [68]. However, our results indicate that the effect may not be the same for all techniques.

3.4 Number of Failing Tests in the Guiding Test Suite

How does the number of failing tests affect the degree to which the generated patches overfit?

Creating test suites with varying number of failing test cases: To measure the effect of number of failing tests on G&V techniques’ performance, we first create subsets of developer-written training test suites that have varying number of failing tests, we then use these test suite subsets to produce patches. For each defect, we first identify the total number of failing tests ($F_t$) in the developer-written test suite. If the number of failing tests is less than five, we discarded that defect as it cannot be used for performing the desired statistical analysis. Figure 8 shows the frequency distribution of failing tests across the 68 defects for which at least one of the three techniques produced at least one patch. Of these 68 defects, only three defects (Chart 26, Closure 115, and Closure 86) have more than five failing tests. Hence, we could use only these three defects to investigate this research question. We are aware of the threat of validity of the results from this question, given the small number of defects that can be analyzed and how these results might not
Figure 8: Frequency distribution of failing tests over the 68 defects for which at least one patch is produced by at least one of the repair techniques.

Figure 9: Distribution of patch quality obtained using developer-written training test suites containing varying number of failing tests.

generalize to a broader corpus of defects. We are not claiming generalizability in this question. However, we think that having a set of defects with a large number of failing test cases is rare in practice and an interesting use case to analyze more in-depth.

To use a consistent methodology with the previous research question, for each of the 3 defects, we choose five failing test targets: $.2F_t$, $.4F_t$, $.6F_t$, $.8F_t$ and $F_t$. For each target number of failing tests $n$, we attempt to create 5 distinct test suite subsets by incrementally adding a test uniformly at random without replacement until we reach the target $n$. Finally, we measure the association between the patch quality and the number of failing tests in the training test suite using the Pearson correlation coefficient.

Results: While GenProg and TRP AutoRepair are able to patch all the three defects, PAR was only able to patch one defect (Closure 115). Figure 9 shows the frequency distribution of the patch quality obtained by evaluating the patches using evaluation test suite. We find that all the patches generated using the three techniques pass at least 96.5% of the evaluation test suite.

Pearson correlation coefficient (0.01 for Chart 26 and 0.05 for Closure 115 using GenProg, 0.08 for Chart 26 and 0.65 for Closure 86 using TRP AutoRepair) implies a very weak correlation. It is worth noticing that our analysis is limited by having only three defects to answer this research question and this defect set might not be representative of a large
sample of test suites with varying number of failing tests.

The results obtained show that, as hypothesized, enhancing key characteristics of the guiding test suite can lead to an improvement of the produced patches. Concretely, this research thrust shows that guiding test suite size and provenance are strong indicators for produced patch quality, and therefore enhancing these components in automatic program repair leads to higher quality plausible patches.

4 Research Thrust II: Analyzing Developer Behavior to Inform APR Selection Mechanisms

A key component in the automatic program repair process is the selection mechanism APR approaches use to choose which edits will be applied to faulty locations when creating patch candidates. This selection is particularly difficult because the search space of possible edits that can be applied to each location is trivially infinite. Most current state-of-the-art APR techniques use equally-distributed or heuristic-based probabilities to choose the edits to be performed in a selected location of the source code \[39, 30, 75, 62, 42, 63\].

In this research thrust, we hypothesize that mimicking human behavior selection decisions when patching software instead of using the current heuristic-based approach leads to higher quality patches (e.g., it is much more common for a developer when fixing a bug to modify the condition within an if statement than to create a new type declaration statement). The heuristics used by state-of-the-art approaches are mostly based in general approximations of reasonable behavior and have not been carefully calibrated. Our 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 the following sections of this research thrust we analyze the behavior of developers when fixing bugs by inspecting the types of statements they modify and the edits they perform to the source code (Section 4.1). We then mine a corpus of popular open source projects and create an empirical probabilistic model of the edits developers use (Section 4.2). Finally we create an APR approach which uses this probabilistic model to select which transformations to apply when creating patch candidates (Section 4.3).

4.1 Analysis of Developer Behavior in Java Projects

Do real-world developers edit certain statement kinds more frequently than others in the bug fixing process?

Using Boa to analyze human behavior: To answer this question, we use the Boa framework \[17\] to query 4,590,679 bug fixing commits from a database of 554,864 Java projects. Bug fixing commits are identified by the Boa framework using the isfixingrevision function
which uses a list of regular expressions to match against the revision’s log [17]. The Java language specification classifies statements into statement kinds (e.g., For Loop, While Loop, Variable Declaration, Assignment, etc.). Since our intuition is that APR can benefit from mimicking the edit behavior of developers, we start by analyzing how developers apply common APR coarse-grain edits (replacement, deletion, and insertion of statements) to each statement kind.

**Results:** We found that the statement kind that most commonly replaces other kinds (replacer) is the *If Statement* (101,366 appearances), and the least common replacer is the *Type Declaration* (447 appearances). The statement kind that is most commonly replaced by other statement kinds (replacee) is the *Return Statement* (111,938 appearances), and the least common replacee was again the *Type Declaration* (399 appearances). These results support our intuition that human developers use some statement kinds more frequently than others.

Regarding deletions and insertions, the most commonly added statement kind is *Expression Statement*, added in 25.7% of the studied cases, followed by *If Statement* (17.2%). The least added was the *Type Declaration*, (0.2%). The most commonly deleted statement kinds are *Expression Statement*, deleted 13.6% of the cases studied; and the least deleted statement kind is the *Type Declaration* (0.2%).

We also used this corpus of bug fixing commits to analyze behavior that is common in automatic program repair, in particular, the assumption most program repair techniques make about the containment of bugs (that bugs are local to only one or two files from a project). When comparing this assumption to developer behavior from the corpus we found that developers modify a median of 2 files per bug fixing commit, which supports the APR assumption of applying local changes to fix bugs.

APR approaches usually have a limited ability to create new code when fixing an error. In this study we also analyzed how often Java developers introduce new code (classes, methods, variables, etc.) in bug fixing commits. On average, developers create 0.16 classes, 0.69 methods (including test methods), and 0.20 variables per bug fixing commit. These low values suggest that declaring new classes, methods, and variables when repairing software is rare and therefore more accessible to APR approaches. This work has been completed and published [70].

### 4.2 Corpus Mining from Popular Github Projects

In Section 4.1 we validated our intuition that when fixing bugs, a set of statement kinds is used more often than others. We then performed an analysis to understand the distribution of mutation operators (types of edits) used by developers when fixing bugs. Therefore our next research question is:

**What is the distribution of edit operations applied by human developers when repairing errors in real world projects?**

**Mining GitHub to build a probabilistic model:** We mined a substantial corpus of bug fixing commits created by human developers from the 500 most popular Java projects in
<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>Edits found (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append</td>
<td>61.03</td>
</tr>
<tr>
<td>Sequence Exchange</td>
<td>15.76</td>
</tr>
<tr>
<td>Delete</td>
<td>9.10</td>
</tr>
<tr>
<td>Param Replacer</td>
<td>5.93</td>
</tr>
<tr>
<td>Param Add/Rem</td>
<td>3.15</td>
</tr>
<tr>
<td>Expression Repl</td>
<td>1.28</td>
</tr>
<tr>
<td>Method Replacer</td>
<td>1.12</td>
</tr>
</tbody>
</table>

Figure 10: Top mined distribution of mutation operators from most common to least. Mutation operators with an index below 1% are omitted.

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 of the code modified in the bug fixing commits, and matching these differences to the mutation operators state-of-the-art APR approaches use.

**Results:** Figure 10 shows the most common edits used by developers when fixing a bug in the analyzed corpus in order of most common to least common. These results confirm our intuition that human developers use some edits more often than others, and also details in a granular way what is the distribution of edit operators accessible to APR approaches that human developers use when repairing errors. The full list can be found in the publication [69].

Using this mined distribution we then designed and executed an experiment (Section 4.3) to compare the quality of the patches generated using an APR approach based in this distribution against approaches using the well-known heuristic-based distribution.

### 4.3 APR Tool Informed by Human Behavior

Finally, we created an APR approach informed by the distribution of Section 4.2. We augmented a publicly available tool to include the analyzed mutation operators. These mutation operators are classified into two families: Statement-edit (coarse-grain edits) and Template-based (fine-grain edits). The APR technique then creates variants of the source code (also known as patch candidates) by selecting edits based in the distribution from the previous section. We then compare the performance of this approach against state-of-the-art APR approaches. Our following research question is:

How does our model-informed automatic program repair tool compare to the state-of-the-art in APR?

**Comparing our APR tool to state-of-the-art techniques:** In this experiment we used the Defects4J [27] database as a corpus of buggy programs. Because we compare to single-edit techniques, we restrict attention to the subset of the Defects4J bugs with single-line

---

17

2https://github.com/squaresLab/genprog4java
human patches (63 buggy programs). We compare our approach against three state-of-the-art APR approaches: GenProg [39], PAR [30], and TRPAutoRepair [62].

Results: Table 2 shows a comparison between the patches found when using our probabilistic model to guide the selection of mutation operators in the context of APR against the patches found on the described bugs using off-the-shelf state-of-the-art approaches. Column 1 shows the defect ID as labeled by Defects4j, the following two columns show statistics about the held out test suites used to evaluate the quality for the patches produced by the different repair approaches. The remaining columns show the number of patches found and if they generalized (√) or not (×) to the held-out test suite.

From the 19 distinct patches created by our approach, 10 pass all held-out test suite (52.6%); 6.6% of GenProgs patches generalize; 22.2% of TRPAutoRepairs; and 23.1% of PARs patches generalize to the held-out test suite. We conclude that a mutation operator selection mechanism informed by developer behavior creates higher quality patches than its heuristic-based counterpart. Further analysis also established that patches found using the probabilistic model are, in average, generated faster than the ones using an equally-distributed model. This work has been completed and published [69].

The results of this research thrust provide evidence about the increase in patch quality when enhancing the statement selection mechanism when building patch candidates. Patch quality and speed of creation increase when using a probabilistic model based on human behavior.

The techniques proposed in the previous research thrusts show an increase in quality of plausible patches by enhancing key components in APR. However, these technique do not create high quality patches in all cases provided which exhibits a possibility for improvement in the remaining challenges in APR. The next research thrust focuses on the last step of the APR process (creation of plausible patches) and how we can aim to increase the diversity of these patches with the goal of consolidating them to increase patch quality.

5 Research Thrust III: Patch Diversity and Consolidation

In our third research thrust, we hypothesize that consolidating plausible patches leveraging semantic and syntactic diversity leads to higher quality fixes. The intuition behind this idea is that each plausible patch might be a partial solution that fails to cover all possible correct executions of the expected behavior of the program, therefore consolidating several plausible patches might increase the number of correct executions and thus the quality of the overall solution.

Figure 11 shows a graphical description of consolidation. Considering the full set of correct executions of a program Image 1 describes the guiding test suite as a limited discrete subset of all possible correct executions. Image 2 shows how low quality plausible patches implement functionality that covers at least the executions described by the guiding test suite. Image 3 shows how by consolidating low quality patches we can cover a superset of the correct executions of its corresponding low quality plausible patches.

As an example of consolidation, Figure 12 depicts a program that displays temperature in different measurement systems. It receives an integer with the degrees in Faren-
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart # 1</td>
<td>93</td>
<td>74.4%</td>
<td>5</td>
<td>×</td>
<td>6</td>
<td>×</td>
<td>4</td>
<td>×</td>
<td>2</td>
</tr>
<tr>
<td>Closure # 10</td>
<td>472</td>
<td>60.2%</td>
<td>2</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Closure # 18</td>
<td>106</td>
<td>73.4%</td>
<td>1</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Closure # 86</td>
<td>435</td>
<td>64.8%</td>
<td>2</td>
<td>✓</td>
<td>–</td>
<td>1</td>
<td>✓</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Lang # 33</td>
<td>85</td>
<td>92.3%</td>
<td>1</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Math # 2</td>
<td>13</td>
<td>100.0%</td>
<td>1</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>✓</td>
</tr>
<tr>
<td>Math # 75</td>
<td>25</td>
<td>92.5%</td>
<td>1</td>
<td>✓</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>Math # 85</td>
<td>11</td>
<td>94.7%</td>
<td>4</td>
<td>×</td>
<td>8</td>
<td>×</td>
<td>3</td>
<td>×</td>
<td>8</td>
</tr>
<tr>
<td>Time # 19</td>
<td>55</td>
<td>86.0%</td>
<td>2</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>1</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 2: Comparison of patches generated using the probabilistic model-based repair and other state-of-the-art approaches. “Held-out tests” and “Line Coverage” represent the number of tests in the held-out test suite and the percentage of covered lines in the modified class of each corresponding defect. “–” indicates no patch found. The “Found” column indicates the number of patches found per bug over the multiple random trials. “Gen?” indicates whether all produced patches generalize to the held-out test suites (✓) or not (×). In these results, all produced patches for a bug, technique pair either generalized, or not.

1. Guiding Test Suite
2. Low Quality Patches
3. Consolidated Patch

Figure 11: Consolidated patch can cover a superset of the correct executions of its corresponding low quality plausible patches

This program contains an error in the case of converting to Celsius (‘c’) where the variable degrees is converted to the correct number, but it is never added to the return string “ret”. This causes Test5 in the test suite to fail showing the erroneous behavior. The expected return value is ”0 °C”, and the actual return value is ”°C”.

heit and a character for the measurement system to use. The return value is a string with the correct degrees and the measurement system. Below we display a test suite used to evaluate its performance.
/* Input:
   * int degrees: degrees in fahrenheit
   * char system: 'c' for Celsius, 'f' for Fahrenheit, 'k' for Kelvin
   * Output:
   * String: detailing the converted temperature and the system used
   * Empty string if invalid system */
public String convertedTemp(int degrees, char system){
    String ret = "";
    if(system == 'f'){
        ret = Integer.toString(degrees);
        ret += "°F";
    } else if(system == 'c'){
        degrees = (int)((degrees - 32) * 5/9);
        ret = "°C"; // bug! "degrees" is never added to "ret"
    } else if(system == 'k'){
        degrees = (int)((degrees - 32) * 5/9 + 273.15);
        ret = Integer.toString(degrees);
        ret += "°K";
    }
    return ret;
}

Test1: assertEquals(convertedTemp(-87,'f'), "-87 °F")
Test2: assertEquals(convertedTemp(55,'f'), "55 °F")
Test3: assertEquals(convertedTemp(2,'k'), "257 °K")
Test4: assertEquals(convertedTemp(-23,'k'), "243 °K")
Test5: assertEquals(convertedTemp(32,'c'), "0 °C")

Figure 12: Illustrative example of a program that displays temperature in different systems

Table 3 shows three different overfitting plausible patches which are able to pass all the test cases in the guiding test suite, but fail to generalize to an independent evaluation. On the bottom an example shows how when these low quality overfitting patches can be consolidated, the resulting patch is able to intuitively generalize to a superset of correct executions. This example shows a simple case where patch consolidation can be used to increase the quality of overfitting plausible patches.

5.1 Preliminary Work

We have conducted an initial experiment showing the potential of patch consolidation using the corpus of plausible patches described in Section 3.1 by consolidating the generated plausible patches using the off-the-shelf software merging tool JDime in the patches generated using three different APR approaches (GenProg, TRPAutoRepair, and PAR).

First, we create all possible combinations of two patches using two merging mechanisms provided by JDime. These merging techniques provide insight about what is the best way to merge patches to get high quality fixes:

- Line-based: syntactic and language agnostic merge based in line comparison between two files. This approach is similar to what is used by the diff command

https://www.unix.com/man-page/all/1/diff/
Samples of three low quality plausible patches that overfit to the guiding test suite

Consolidated patch that generalizes to an independent evaluation

Table 3: Example of three overfitting plausible patches which, when consolidated, generalizes to an independent evaluation

and GitHub merge.

- Structured: language dependent merge based in the abstract syntax tree structure of the program. This approach is much more resource intensive than line-based.

We then used held-out test suites to evaluate the quality of the consolidated patches (Section 2). Figure 13 describes the behavior of the consolidated patches. Up to 53% of the consolidated patches show higher quality than at least one of their corresponding individual plausible patches, and up to 36% 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. The following sections describe proposed work for this research thrust:

5.2 Improve Diversity of Generated Patches

In the population of plausible patches we have generated, it is common to find patches that are semantically identical. This phenomenon is presented when one program can be replaced with another without affecting the observable results of executing the program [61] (programs have the same logical content [50]). Figure 14 shows three plausible patches GenProg found (Closure 13). All of them pass all test cases in the guiding test suite.

The first patch performs the minimum change necessary for the failing test case to pass: removing line 49. The second plausible patch deletes line 49, but also adds an irrelevant line of code (line 50). Finally, the third patch performs similar changes to the
previous patch, instead of adding a repeated line of code, it adds an if statement that is not executed by the guiding test cases. In examples as such where there is little or no semantic diversity, consolidating these patches would not lead to an improvement in quality given that functionality would only be repeated or overwritten.

Our patch consolidation technique would benefit from incentivizing these approaches to look for patch diversity to create more diverse plausible patches. We propose to enhance the methodology to traverse the search space of patches candidates so that the plausible patches found are more likely to be semantically distinct, therefore amplifying the potential of increasing quality by patch consolidation. The research questions to be answered regarding this topic are the following:

- Does the quality of plausible patches increase by incentivizing diversity in the automatic program repair process?
- Do patches created by our diversity-driven approach increase quality when consolidated?
- How does the quality of diversity-driven consolidated patches compare to the quality of the non diversity-driven consolidated patches?
5.2.1 Modify the Fitness Function and Implement Multi-Objective Search 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. Most current approaches set their fitness function to look mainly for patch correctness (e.g., [39, 75, 69] 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 optimize for both patch correctness and semantic diversity. We propose multi-objective search as opposed to the current one-dimensional approach. We require a quantitative measurement to encode it into the multi-objective search criteria. Since program equivalence is undecidable [66], we have proposed a semantic measurement to approximate software equivalence and its trivial opposite, software diversity in Section 5.2.2.

5.2.2 Proposed Test-Suite-Based Semantic Difference Measurement

Optimizing for patch diversity requires a quantitative measurement of how different a program is with respect to another. Figure 15 diagrams the proposed process to approximate the semantic difference between two programs: Program A and Program B. The first step is to create a specification from each of the programs in the form of a test suite describing the behavior of the program as a set of inputs and expected outputs. This can be achieved using test suite generation tools (e.g., Evosuite [20], Randoop [55]).

The second step is to combine both specifications into a single Specification AB. This consolidated specification is used to evaluate each individual program (Step 3). We then compare the reports from the evaluations using Hamming distance (Step 4) to identify which cases behave differently. Hamming distance is fast and assumes equal distance strings, making it appropriate for our analysis. Finally, we compute the semantic differ-
ence based in the Hamming distance as a percentage of the number of test cases (Step 5). As a result we obtain a metric describing how semantically different is program A from program B.

It is possible that the joint test suite is not able to compile due to the lack of functionality in one of the programs (e.g., if program A does not have a method evaluated by the test suite generated from program B, the joint test suite will not be able to compile to evaluate program A). In this scenario, this test case behaves differently in both programs and we annotate it accordingly when computing the Hamming distance value. By definition, absent functionality from a program can not behave equally to present functionality in another program. This behavior is unlikely to happen in modern APR approaches.

5.2.3 Slicing Mutation Operators, Fault Space, or Test Cases

To mitigate the risk that the multi-objective search approach might be ineffective, I propose three alternative mechanisms to improve diversity that I will explore only if necessary. A 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. In a scenario where the previous multi-objective function does not create more diverse patches, we propose to restrict the search space in three different ways: Slicing mutation operators, fault locations, or test cases.

- When slicing mutation operators we 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, the approaches will be forced to look for candidate patches considering only a set of non-overlapping fault locations.

Figure 15: Diagram describing the proposed semantic difference measurement.
• When slicing test cases, we will have the APR approaches consider only a set of the failing test cases for the buggy programs that include more than one negative test case. This will create patches that are able to pass the analyzed set of negative test cases, and when consolidated, the solution might comprehend the behavior described by all failing test cases.

5.3 Analyzing Meaningful Characteristics of Consolidated Patches

After the experiment described above, we are later interested in understanding what characteristics of consolidated patches correlate with high and low patch quality. The goal is to learn these characteristics to be able to create proxies for high quality consolidated patches and encourage the use of these attributes in the creation of future consolidated patches.

To analyze these characteristics we plan to compare the syntactic changes of the created high quality consolidated patches to their counterpart: low quality consolidated patches. We also plan to compare them to their corresponding lower quality individual patches, and to their corresponding human patch. Research questions to be answered in this section are the following:

• Do smaller number of changes correlate with higher quality?
• Is there a particular kind of change that correlates with higher quality?
• Are high quality consolidated patches more similar to the human patch than lower quality consolidated patches?

To achieve this experiment, we need a corpus of patches with their corresponding quality and a mechanism to understand the differences between these sets of patches. These are described below:

5.3.1 Obtaining Patch Quality Corpus

To evaluate the previously described concepts, we plan to measure the quality of the diverse patches generated and their corresponding consolidated patches. We will use the multi-objective fitness function methodology described in Section 5.2.1 to generate the patches and then use the methodology described in Section 5.1 to merge the diverse plausible patches into consolidated patches. Finally, we will create held-out test suites based in the oracle description of correct patch generated by the human developers to evaluate the quality of the diverse plausible patches generated by the APR approach and the quality of the combined consolidated patches.

5.3.2 AST-Based Syntactic Difference

To answer the research questions described above we need a way to compare the changes performed when creating consolidated patches. A common way to analyze the syntax of software is by using abstract syntax trees (AST). Figure 16 shows an edit in source...
code and its corresponding before-edit AST and after-edit AST. Tree differencing provides information not only about how many lines were modified, but it also provides syntactic information about the sections being modified. In this example AST differencing can tell us that the line modified was within a block, as part of a while loop, that it is an assignment, and it is substracting a variable and a constant, therefore making the AST-Based syntactic difference a more fine-grained measurement.

<table>
<thead>
<tr>
<th>Code</th>
<th>AST Before Edit</th>
<th>AST After Edit</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>y = a + b;</code></td>
<td><img src="image1.png" alt="AST Diagram" /></td>
<td><img src="image2.png" alt="AST Diagram" /></td>
</tr>
<tr>
<td><code>while(y &gt; a){</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>a = a + 1;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>- b = b - 1; }</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 16: Abstract syntax tree representation of a code modification. The left cell shows the code being modified, the center cell shows the AST before the edit is performed, the right cell presents the AST after the edit was implemented.

We can apply tree differencing algorithms to analyze the changes performed when creating higher quality consolidated patches and others (low quality consolidated patches, single patches, and the human patch) using tree edit distance [9, 24]. This differencing technique will provide a quantitative measurement to be able to answer the proposed research questions. Tools in this realm include Gumtree [18], JDime [3] or APTED [57].

By promoting diversity in the APR process and consolidating plausible patches into higher quality fixes we are enhancing the last step of the APR process leading to an increase in patch quality. The rest of this proposal describes the tasks to completion (Section 6.1), timeline (Section 17), risks (Section 6.3), and related work (Section 7).

6 Research Plan

6.1 Tasks to Completion

The proposed tasks for completion are detailed as follows:

- **Create APR Tool to Incentivice Diversity:**
  We plan to modify the fitness function in the APR process and implementing multi-objective search to optimize for diversity and correctness.
• **Run Experiment to Obtain Diverse Patches:**
  We will run an experiment whose primary purpose is to obtain a more diverse pool of patches in comparison with previously analyzed semantically and syntactically similar patches.

• **Consolidate Individual Patches:**
  Using state-of-the-art code merging approaches we will consolidate diverse patches for the same bug.

• **Create Held-Out Test Suites and Evaluate Plausible Patch Quality:**
  We can then create held-out test suites based in the oracle solution provided by Defects4J as a quality assessment mechanism to evaluate the quality of the initial diverse pool of patches and the generated consolidated patches.

• **Analyze Results:**
  Using the proposed measurements for diversity we can analyze how dissimilar are the patches generated using our diversity-driven approach in comparison to the patches generated using the one dimensional fitness function, and after consolidation we can analyze the quality of the consolidated patches from both provenances (diversity-driven and state-of-the-art).

• **Prepare Thesis Defense:**
  Write the document and prepare the presentation.

### 6.2 Timeline

The proposed timeline shown in Figure[17] includes the two most relevant previous projects and the ongoing current project. The last portion covers 2 years and 2 months of work, starting with the preliminary work and preparation of this proposal, and finishing with the thesis defense. The total span of the work described in this proposal is 4 years and 6 months.

### 6.3 Risks

Some potential risks of the future work of this thesis are the following:

**Risk: It is difficult to find plausible patches when implementing multi-objective search and modifying the fitness function**

There is a possibility that when implementing multi-objective search and modifying the fitness function of the APR process it will be more difficult to find patches due to the fact that it will not be looking solely for correctness of patches but also for diversity. Therefore the diversity factor might deviate the fitness function to parts of the search space where it will be more difficult to find a solution.

A small number of patches is not a problem as long as the patches found become high quality patches once consolidated. But if we find a smaller number of patches that do not necessarily improve quality when consolidated, our immediate fallback option is to
Figure 17: This diagram describes the proposed timeline for the dissertation. Each vertical straight line describes one year of work starting from June 2016 to December 2020. The distance between the straight and the dotted lines represents a 6 month period.

run a second experiment using mutation operators, fault location, or test case slicing as described in Section 5.2.3.

Risk: The proposed semantic difference measures do not capture inequality at the desired granularity level
It is possible that, in some cases, the specifications generated from the programs do not capture parts of the semantic and syntactic space that distinguish one patch from another. To address this, we can use the methodology described in Figure 15 but capturing the specification of the program using different methodology such as invariant detection.

Risk: Implementing multi-objective search and modifying the fitness function does not create more diverse patches
It is possible that modifying the fitness function does not lead to more patch diversity and our experiments create patches with similar diversity than with the state-of-the-art fitness function. If this is the case, besides learning more about how a multi-objective fitness function behaves in the context of automatic program repair, we still have interesting research novelties in this study independently of the diversity of the patches found such as the proxy measurements to measure program diversity and how patch consolidation can be used to increase patch quality in patches generated with state-of-the-art repair approaches.

Risk: Consolidating patches does not lead to higher quality fixes
There are two risk scenarios: Patch consolidation leads to high quality fixes only in a small portion of the analyzed cases, and patch consolidation does not lead to high quality fixes in any case. For the former scenario, we would have a smaller corpus of analysis but we would still be able to carry out the proposed work although the results might not generalize as much as we intent to. For the latter scenario, even though it is extremely
unlikely to have zero cases of improved quality, we would still have learned about patch diversity in the context of APR and how multi-objective search influences the quality of plausible patches found.

7 Related Work

There have been previous efforts to increase the quality of candidate patches in automatic program repair.

7.1 Search-Based Approaches

There are previous program repair techniques that, similar to our approach, target the Java programming language. PAR [30], for example, searches for common patterns used by developers to generate fix templates. These templates are single-edit modifications of the source code that are used often to fix bugs. ARJA [80] is a Java-focused repair technique that implements multi-objective search and uses NSGA-II to look for simpler repairs. It also introduces a test filtering procedure that can speed up the fitness evaluation of GP and three types of rules that can be applied to avoid unnecessary manipulations of the code. Genesis [43] is a repair approach that processes human patches to automatically infer code transforms for automatic patch generation. The authors use patches and defects collected from 372 Java projects. QACrashFix [22] is a repair approach that extracts fix edit scripts from StackOverflow and attempts to repair programs based on them. Part of this proposal uses certain functionality from QACrashFix to account for replacements in the human behavior study (Section 4.2).

Previous tools have also tried to modify the objective function of APR approaches following different methodologies from ours. HDRepair [35] 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 [42] uses a probabilistic model built on the history of 8 different projects to rank candidate patches. It learns model parameters via maximum likelihood estimation. Unlike this work, we apply mined knowledge when actually instantiating patch candidates rather to rank the already created patch candidates, which reduces the search space at creation time.

GenProg [39], PAR [30], and TRPAutoRepair [62] are examples among a family of syntactic-based automatic program repair approaches seeking to generate patch candidates by modifying program syntax (while semantic-based approaches use code synthesis to construct fix code). These repair approaches represent a variety of search-based techniques that vary in mutation operator kind and search traversal, therefore we compare against them in this proposal. GenProg [39] is an APR approach that leverages genetic programming while modifying software syntax using coarse-grained mutation operators such as delete, append, and replace. TRPAutoRepair [62, 63] traverses the search space using random search and restricts its pool of possible patches by applying a single edits to its candidate patches. The authors of this approach evaluate their tool against Genprog and suggest that TRPAutoRepair outperforms GenProg in 24 bug benchmark. PAR [30]
uses a set of templates mined from human behavior to modify source code (e.g., check if a variable is null before using it). 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 [68].

There are repair techniques that resemble or extend the approaches we compare against. Similar to PAR, LASE [51] learns edit scripts from a pool of bug fixing examples, finds the appropriate edit locations and applies the customized edit to the selected location. The scripts consist in a sequence of operations (insert, delete, update, and move) applied to the nodes of an abstract syntax tree representation of the program. These scripts are learned from examples changed in syntactically similar ways. SPR [41] combines staged program repair and condition synthesis to find repairs in programs. This repair approach introduces a set of parameterized transformation schemas to generate and search a diverse space of program repairs. Their evaluation in 69 bugs from 8 open source programs indicate an improvement over previous approaches.

Several techniques also resemble or extend GenProg, for example, AE [75] uses adaptive search to improve the order in which test cases and candidate patches are evaluated to improve the usage of resources in the APR process. Because of this prioritization technique, AE reduces the search space size by an order of magnitude as compared to GenProg. JAFF [4, 5] is a repair technique based on co-evolution where programs and test cases co-evolve together. These components influence each other with the aim of fixing errors in programs automatically. Kali [64] is a tool that focuses in the removal of source code statements as a means to pass all test cases from a test suite.

Other known repair techniques that target domain-specific defects are: LeakFix [21] is a safe memory-leak fixing tool for C programs. It uses pointer analysis to build procedure summaries. Based on these, it generates fixes by freeing memory in key program points. Schlute et al. [67] developed and automatic program repair system for arbitrary software defects in embedded systems. It targets mostly systems with limited memory, disk and CPU capacities. It does not require the source code.

7.2 Semantic-Based Techniques

Different from the work performed in this proposal, another big family of approaches are semantic-based techniques, which use semantic analysis to construct candidate patches [38, 48, 49, 54, 28]. Similarly, synthesis-based repair is a family of techniques that uses constraints to build patches following the constraint’s description. These constraints may take the form of developer-generated specifications, formal verification, invariants, among others [25, 59]. Such techniques typically use synthesis to construct repairs, using a different mechanism than our approach for both constructing and traversing the search space, therefore our approach is less immediately comparable. Some examples of this family of approaches are SemFix [54], DirectFix [48], QLOSE [15], Angelix [49], S3 [38], ACS [77], and Nopol [78].

SemFix [54] generates repair constraints using symbolic execution and the guiding test suite, it then solves the constraints using an SMT solver. DirectFix [48] uses partial maximum SMT constraint solving and component-based program synthesis building simpler and safer patches than SemFix. Angelix [49] focuses on a repair constraint to reduce
the search space named “angelic forest” independent of program size, which represents a considerable improvement in scalability over its predecessors [54, 48]. Recently a Java version was proposed called JFix [37]. QLOSE [15] is an approach which finds plausible patches by minimizing an objective function based on semantic and syntactic distances from the buggy version. S3 [38] focuses on a programming-by-examples methodology which uses code synthesis to find plausible patches. ACS [77] targets if conditions, using dependency-based ordering and predicate mining. Nopol [78] is a Java-focused approach which targets if conditions. It uses an SMT solver and angelix fix localization to create plausible patches for the buggy programs. Part of the work performed in this proposal was evaluated against Nopol. The original publication does include this comparison [69]. However, for the purposes of this proposal we consider these results to be out of scope, comparing our search based approach against a semantic based approach.

7.3 Software Diversity

Similar to our approach, there have been previous attempts to improve the quality of software by incentivizing diversity. One of the biggest motivations in this direction is N-Version Software (NVS), which is a way to take advantage of different implementations of code created following the same specification [8]. It was first introduced in 1977 as the independent generation of \( N \geq 2 \) functionally equivalent programs from the same initial specification [6]. One of the major justifications for NVS was that it would be able to provide online tolerance for software faults, following the intuition that the independence of programming efforts will reduce the probability of identical software fault behavior.

Some key experimental hands-on studies that have researched NVS are, for example, Avizienis [6] and Chen [12], where they implemented NVS systems using 27 and 16 independently written versions; Ram [65] and Vog [73] have studied real-time software by developing six different implementations (programming languages) from the same requirements. Diversity in this context usually refers to the diversity of components (e.g., different compilers, programming languages, versions of the specifications [29], or different algorithms [12]). Even when software diversity is enforced through the variation of programming languages, developers tend to follow a “natural” sequence even when coding independent computations that could be performed in any order [7]. In this thesis proposal, we avoid having this restriction since our patches are not generated by human developers and therefore do not follow any sequence that may seem “natural” to human programmers.

Another impediment to encouraging software diversity in software systems is when software specifications describe ‘how’ to implement portions of the code [7]. Because of this, identical errors were found in various versions. In this proposal, our specification is given by test cases, which describe examples of correct behavior. APR tools do not follow any instructions on “how” to build the patch, and the nature of the specification varies, removing this impediment human developers have. Further research has advanced in relationship to software diversity metrics in the context of NVS [11, 44]. These studies focus mostly in the diversity of fault behavior. Robustness of software has also been analyzed in the context of operating systems using similar diversity metrics [32]. In this proposal, we are not interested in focusing on fault behavior among versions but we are
interested in creating diverse patches for the same bug.

More recently, artificial diversity has been proposed to improve the correct location of errors in software using “Mutation-Based Fault Localization” (MBFL) approaches \cite{52,56}. The intuition behind these techniques is that when mutants are generated at the faulty location, the test suite should exhibit different behavior than when mutants are generated in non-faulty locations. Further studies \cite{72,58} have suggested that MBFL techniques do not significantly distinguish between faulty and non-faulty locations. Smith et al. \cite{68} 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. This paper introduces the usage of a held-out independently created test suite as a means for measuring software quality. The work introduced in this proposal leverages these previous ideas to create real N-version patches with actual compilable code to be executed by the test cases.

8 Conclusion

A fundamental problem current automatic program repair approaches suffer is the generation of low-quality patches that overfit to the guiding test suite and do not generalize to an oracle evaluation. This thesis proposal describes a set of mechanisms to enhance key components of the automatic program repair process to generate higher quality patches, including an analysis of test suite behavior and how their key characteristics improve the creation of plausible patches in automatic program repair, an analysis of developer behavior to inform the mutation operator selection distribution and an statement kind selection, and a repair technique based on a multi-objective search space traversal to increase patch diversity as a means to create consolidated higher quality fixes.

References


