6-2018

Overfitting in automated program repair: Challenges and solutions

Dinh Xuan Bach LE
Singapore Management University, dxb.le.2013@phdis.smu.edu.sg

Overfitting in Automated Program Repair: Challenges and Solutions

LE DINH XUAN BACH

SINGAPORE MANAGEMENT UNIVERSITY

2018

Overfitting in Automated Program Repair: Challenges and Solutions

by

Le Dinh Xuan Bach

Submitted to School of Information Systems in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in Information Systems.

Singapore Management University
2018

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to deeply thank my supervisor, David Lo, whom I considered myself extremely lucky to work with. Under his guidance, I have leant to grow both personally and professionally. Without his kind advice and constant help, this wonderful journey would have not been possible for me.

I would also like to thank Claire Le Goues, Duc-Hiep Chu, Quang-Loc Le, Willem Visser, Wei-Ngan Chin, Julia Lawall, Quyet-Thang Huynh and many other friends and colleagues for their great support for my research works and career.

I am also grateful to the School of Information Systems at Singapore Management University (SMU), for providing me generous scholarship to pursue studies at SMU and Carnegie Mellon University (CMU).

Lastly, I would like to thank my family for all their love and encouragement. For my grandparents, parents, and sister who always supported me in all my pursuits. For my wife and son (Quynh-Nga Pham and Dinh-Quan Le) who always encouraged me with their unconditional love.

Chapter 1

Introduction

This chapter discusses the main problem and motivation of this dissertation. It also discusses a quantification of various research issues directly related to the dissertation. A summary of works done will also be presented along with the structure of the dissertation.

1.1 Software Bugs & Motivations

1.1.1 Manual Bug Fixing & Its Cost

Software bugs are one of the primary challenges in software development, which usually significantly diminish software quality. A software bug is a problem causing a program to crash or produce invalid output. A bug can be an error, mistake, defect or fault, which may cause deviation from expected results, an explicit security vulnerability that may be maliciously exploited, or a service failure of any kind [8].

Bug fixing is notoriously difficult, time-consuming, and requires much manual efforts. U.S. National Institute of Standards and Technology reported that software bugs were estimated to cost the U.S. economy more than 50 billions of dollars annually [84]. Given short time to market, mature commercial software systems are often delivered with both known and unknown bugs, despite the support of multiple developers and testers dedicated for such projects [55, 6].

Fixing bugs has become a dominant practical concern, given the prevalence and cost of software bugs, much of which is often attributed to the fact that state of the art in bug-fixing practice still relies on human to manually fix them. To correctly fix a bug, human developers have to understand the intended behaviors of the system implementation, and why the current implementation does not follow the intended behaviors. Based on that understanding, human developers have to figure out what changes to the program source code or configuration will correct the buggy implementation. Often, those changes (regarded as fix or patch) will be validated (e.g., through a code review) before being committed or deployed to the users.

Manually fixing bugs is costly in terms of lost productivity and money to pay developers to fix bugs [91], and this cost can increase by orders of magnitude as development progresses [93]. This, in fact, leaves many defects, including security-critical defects, unaddressed for extensive periods [31]. Thus, there is a dire need to develop automated solutions that help mitigate the onerous burden on manual human bug fixing, by automatically and generically repairing variety of bugs from large real-world software systems.

1.1.2 Automated Program Repair & Its Challenges

Previous subsection highlights the fact that bug fixing is time-consuming and costly in practice, which rests entirely on manual human efforts. Thus, automated program repair (APR) techniques that can automatically repair software bugs in the wild would be of tremendous value.

Overview of APR. Substantial recent works on APR have been proposed to repair real-world software, making the once-futuristic idea of APR become gradually materialized. These repair techniques generally fall into two categories: search-based methodology (e.g., *GenProg* [52], *PAR* [37], *SPR* [58], *Prophet* [60]) and semantics-based methodology (e.g., *SemFix* [68], *Nopol* [96], *DirectFix* [65], and *Angelix* [66]). Search-based repair techniques generate a large pool of possible repair candidates, i.e., search space of repairs, and then search for correct repair within that search space using an optimization function. Meanwhile, semantics-based techniques leverage constraint solving and program synthesis to generate repairs that satisfy semantics constraints extracted via symbolic execution and provided test suites.

Challenges. Proposed repair techniques in both families, despite varying in the ways they search for repairs, rely heavily on test cases to guide the repair process and validate patches – a patch is deemed as *correct* if it passes all tests used for repair (aka. *repair test suite*) [52]. While test cases are commonly used in practice to guard against unexpected behavior of software, they are known to be incomplete and often weak. Thus, machine-generated patches, which pass all tests, may still be indeed judged incorrect. This is often regarded as *patch overfitting* [81], in which machine-generated patches overfit to the repair test suite, but do not necessarily generalize to other test sets or desired behavior that developers would expect. It has been shown that patches generated by *GenProg* and its early search-based counterparts are often incorrect although they pass all tests [74], and that the degree to which search-based APR techniques suffer from overfitting is high [81]. Patch overfitting has thus progressively been a pressing concern in APR. To address the overfitting problem, several challenges need to be overcome.

Challenge 1: Scalability and tractability. The ultimate practical goal of APR is to cheaply scale to various large, realistic software systems, and yet, to be able to produce correct repairs for those systems. However, tractability often comes with the territory of being scalable. That is, it has been shown that the search space for repairs generated by APR techniques is often huge, in which plausible repairs – which pass all tests but are incorrect – are dominant [59]. This poses significant challenges on finding correct patches among the huge search space. Thus, to achieve its goal, APR techniques must be able to *efficiently* and *effectively* manage as well as navigate the search space to find correct patches.

Challenge 2: Expressive power. Current state-of-the-art APR techniques still cannot correctly fix many bugs from various real-world software. For example, GenProg generates patches for 55 bugs, out of which only fewer than three patches are correct [74]. Yet, these bugs only come from a few software systems. Thus, there is a need to enhance the expressive power of APR to generate correct patches for many more bugs from variety of real-world programs in practice.

Challenge 3: Patch validation. Overfitting is not only attributed to the ways APR techniques manage and navigate the search space, but also the methodologies used to validate automatically-generated patches. Early methodology leverages only repair test suite for patch validation – a patch is deemed *correct* if it passes the repair test suite, and *incorrect* otherwise. This method, however, has recently been shown ineffective – majority of patches that pass the repair test suite are indeed still judged *incorrect* [81, 59]. This motivates new methodologies for patch validation, which rely on other criteria rather than repair test suite alone. Recent works adopt the following two methods separately:

- **Automated annotation by independent test suite.** Independent test suites obtained via a automatic test case generation tool are used to determine correctness label of a patch – see for example [81, 49]. Following this method, a patch is deemed as *correct* or *generalizable* if it passes both the repair and independent test suites, and *incorrect* otherwise.

- **Author annotation.** Authors of APR techniques manually check correctness labels of patches generated by their own and competing tools – see for example [95, 56]. Following this method, a patch is deemed as *correct* if authors perceive semantic equivalence between generated patches and original developer patches.

While the former is incomplete, in the sense that it fails to prove that a patch is actually correct, the latter is prone to author bias. In fact, these inherent disadvantages of the methods have caused an on-going debate as to which method is better for assessing the effectiveness of various APR techniques being proposed recently. Unfortunately, there has been no extensive study that objectively assesses the two patch validation methods and provides insights into how the evaluation of APR's effectiveness should be conducted in the future.

Put simply: patch overfitting has become an important challenge in APR. To avoid overfitting, there is a dire need to both improve the APR techniques themselves in the way they manage and navigate the search space, and provide insightful guidelines on how the effectiveness of APR techniques should be evaluated.

1.1.3 Works Completed

This dissertation tackles the challenges in the overfitting problem of APR in various angles, seeking to provide insights and solutions that help push the boundaries of both search- and semantics-based APR further. At the point this dissertation is written, four pieces of work have been completed, of which there have been published and the remaining work is under revision for submission. The three published works improve both search- and semantics-based APR, and empirically study the overfitting in APR. The latter work – under revision – evaluates the reliability of methodologies used to validate machine-generated patches.

In particular, we first propose *HDRepair* [50] – a search-based APR system that leverages bug fixes submitted by developers in the history of many large, real-world software to effectively and efficiently guide the bug fixing process. Second, we show that semantics-based APR techniques, similar to its search-based counterparts, also suffer from a high degree of overfitting [51]. We do so by implementing a semantics-based APR framework on top of *Angelix* [66] that enables many different synthesis engines to be used for synthesizing repairs [49], and studying various characteristics of the synthesis engines in the context of APR. Third, we propose *S3* [46] – a scalable semantics-based repair synthesis engine that is capable of synthesizing *generalizable* repairs, which mimic repairs submitted by developers, by leveraging programming by examples methodology [27]. Last, we propose to empirically study the effectiveness of popular methodologies used for assessing patch correctness, and provide several insights and guidelines for how patches generated by future APR techniques should be evaluated.

1.1.3.1 History Driven Program Repair

The first part of this dissertation presents and evaluates *HDRepair* [50], a search-based APR framework that utilizes the wealth of bug fixes across projects in their development history to effectively guide and drive a program repair process. The main insight for the success of *HDRepair* is that recurring bug fixes are common in real-world applications, and that previously-appearing fix patterns can provide useful guidance to an APR technique.

Like several previous search-based APR methods, *HDRepair* makes use of a stochastic search process to generate and then validate large numbers of patches, seeking one that causes previously-failing tests to pass. In contrast with previous search-based approaches, which by and large use input test cases to assess intermediate patch suitability, *HDRepair* evaluates the fitness or suitability of a candidate patch by assessing the degree to which it is similar to prior bug-fixing patches, taken from a large repository of real patches. Leveraging history of bug fixes helps *HDReppair* in steering clear of patches that overfit to test suite used for repair.

HDRepair displays the following properties:

1. **Scalability:** it cheaply scales to large, real-world Java programs.

2. **High-quality repairs:** it produces human competitive repairs by using

a knowledge base built from software development history as a way to guide and assess patch quality.

3. **Expressive power:** it is able to fix various defects that appear in practice, significantly outperforming state of the art APR.

Extensive experiments on 90 real-world bugs from existing large software systems have been performed to demonstrate the aforementioned properties of *HDRepair*. This work has been accepted for publication in the proceedings of 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) 2016, Research Track.

1.1.3.2 Overfitting in Semantics-based Program Repair

The second part of this dissertation presents an empirical study on the overfitting problem in semantics-based APR. The degree to which a technique produces patches that overfit has been used post factum in recent studies to characterize the limitations and tendencies of search-based APR techniques [81], and to experimentally compare the quality of patches produced by novel search-based APR methods [36]. There is no reason to believe that semantics-based APR is immune to this problem. Unfortunately, there exists no such extensive study in semantics-based APR to date. In this work, we attempt to address this gap.

We comprehensively study overfitting in semantics-based APR. We perform our study on Angelix, a recent state-of-the-art semantics-based APR tool [66], as well as a number of syntax-guided synthesis techniques used for program repair [49]. We evaluate the techniques on a subset of the IntroClass [53] and Codeflaws benchmarks [82], two datasets well-suited for assessing repair quality in APR research.

Overall, we show that overfitting does indeed occur with semantics-based techniques. We characterize the relationship between various factors of interest, such as test suite coverage and provenance, and resulting patch quality. We observe certain relationships that appear consistent with results observed for heuristic techniques, as well as results that are different from those achieved on them. These results complement the existing literature on overfitting in search-based APR, completing the picture on overfitting in APR in general. This is especially important to help future researchers of semantics-based APR to overcome the limitations of test suite guidance. We argue especially (with evidence) that semantics-based program repair should seek stronger or alternative program synthesis techniques to help mitigate overfitting. This work has been accepted for publication at Empirical Software Engineering Journal in November 2017, and for presentation at Journal First Track of International Conference on Software Engineering (ICSE) 2018.

1.1.3.3 Syntax- and Semantic-Guided Repair Synthesis

The third part of this dissertation presents and evaluates *S3* – a semantics-based APR technique that is capable of synthesizing generalizable repairs [46]. As previous subsection highlighted, semantics-based APR, similar to its search-based counterpart, is subject to a high degree of overfitting, motivating the need of stronger repair synthesis system that can generalize beyond the incomplete specifications encoded via test cases.

S3 can generate high quality repairs which are competitive with human-submitted ones. The novelty in *S3* that allows it to tackle the search space to create more general repairs is three-fold: (1) A systematic way to customize and constrain the syntactic search space via a domain-specific language, (2) An efficient enumeration-based search strategy over the constrained search space, and (3) A number of ranking features to rank candidate solutions and prefer those that are more likely to generalize. The ranking functions are guided by the intuition that a correct patch is often syntactically and semantically proximate to the original program, and thus measure such syntactic and semantic distance between a candidate solution and the original buggy program.

Experiment results on 52 bugs from small programs and 100 bugs from large real-world programs show that *S3* is scalable and able to generate generalizable repairs. *S3*'s expressive power and the quality of the patches it generates significantly outperform state-of-the-art baseline techniques (Angelix [66]; and Enumerative [4], and CVC4 [77] two alternative syntax-guided synthesis approaches). This work has been accepted for publication in the proceedings of the 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), 2017, Research Track.

1.1.3.4 Reliability of Patch Correctness Assessment

The fourth part of this dissertation presents an empirical study on reliability of patch correctness assessment methods. Two methods are assessed in this study, including: (1) Automated annotation, in which patches are automatically labeled by using an independent test suite (ITS) – a patch is deemed *correct* or *generalizable* if it passes the ITS, and *incorrect* otherwise, (2) Author annotation, in which authors of APR techniques annotate the correctness labels of patches generated by their and competing tools by themselves.

While automated annotation fails to prove that a patch is actually correct, author annotation is prone to subjectivity. Using either method, there could be potentially wrong judgments on patch correctness, e.g., an overfitting patch – a patch that passes all tests but does not generalize, may be unduly judged as correct. These drawbacks, in fact, have caused an on-going debate on how the effectiveness of an APR technique should be assessed. This has increasingly become an especial concern given the abundance of APR techniques being proposed recently.

To address this concern, we propose to assess reliability of author and automated annotations for patch correctness assessment. We do this by first constructing a gold set of correctness labels for 189 randomly selected patches generated by 8 state-of-the-art APR techniques by means of a user study involving 35 professional developers as independent annotators. By measuring inter-rater agreement as a proxy for annotation quality – as commonly done in the literature – we demonstrate that our gold set is on par with other high-quality gold sets. Through an in-depth comparison of labels generated by author and automated annotations and this gold set, we assess the reliability of the popular patch assessment methodologies. We subsequently report several findings and highlight their implications for future APR studies.

1.1.4 Structure of This Dissertation

Chapter 2 describes related work. Chapter 3 describes *HDRepair* – a search-based history driven program repair framework. Chapter 4 describes the overfitting problem in semantics-based program repair, followed by Chapter 5 which describes *S3* – a semantics-based repair technique. Chapter 6 describes our study on the reliability of patch validation methodologies. Chapter 7 concludes the dissertation.

Chapter 2

Related Work

In this section, we describe related work in automated program repair, including repair techniques, datasets and empirical studies on program repair.

Program repair. Recent years have seen a proliferation in the development of program repair techniques. Repair techniques can generally be divided into two families: search-based vs semantics-based families. Search-based approaches generate a large number of repair candidates and employ search or other heuristics identify correct repairs among them. Semantics-based approaches extract semantics constraints from test suites, and synthesize repairs that satisfy the extracted constraints. GenProg [89, 52] is an early APR tool that uses genetic programming as a search-based to search for a repair that causes a program to pass all provided tests cases among a possibly huge pool of repair candidates. AE [90] leverages an adaptive search approach to search for similar syntactic repairs. [73] propose RSRepair, which uses a random search approach to find repairs, and show that RSRepair is better than GenProg on a subset of GenProg's benchmark. Other recent techniques belong to the search-based repair [58, 60, 50] use condition synthesis, and development history [37] to guide the search for repair. Semantics-based repair approaches [38, 68] typically use symbolic execution and program synthesis to extract semantic constraints and synthesize repairs that satisfy the constraints; other work exists at the intersection of the two families [36]. There further exists work that is farther afield that uses abstract interpretation, unguided by test suites, but requiring specially-written, well-specified code [57]. In a similar vein, for Java, Nopol [96] translates the Object-oriented program repair problem into SMT formulae and uses a constraint solver to synthesize repairs. Our recent work JFIX translates and extends Angelix to work on Java programs, which we adapt to study real-world bugs here [45].

Dataset. Researchers have created a number of benchmarks intended for research and empirical studies in testing, fault localization, and program repair. Defects4J [34] includes more than 300 real-world defects from five popular Java programs. Defects4J is originally intended for to facilitate fault localization research, but it is likely suitable for program repair research as well. The ManyBugs and IntroClass benchmarks [53] provide collections of bugs for C programs, including large real-world C programs, and small C programs as students' homework assignments. The two benchmarks serve different empirical purposes, and are suitable for different types of studies. The IntroClass benchmark contains several hundreds of small C programs, written by students as homework assignments in a freshmen class. Although the IntroClass benchmark only contains small programs, its unique feature is that it includes the two high-coverage test suites: black-box test suite generated by the course instructor, and white-box test suite generated by the automated test generation tool KLEE [13]. This feature makes it suitable for assessing overfitting in automated program repair; one test suite can be used for repair, and the remaining test suite can be used for assessing the quality of generated patches. Recently, Codeflaws was proposed as another benchmark for assessing automatic repair techniques following the spirit of the IntroClass benchmark. Codeflaws contains 3,902 defects from 7,436 small programs from programming contests hosted by Codeforces,[1] each of which contains two independent test suites.

[1]http://codeforces.com/

Empirical Studies on Program Repair. The rapid growth of program repair techniques has motivated empirical studies that compare and reveal strengths and weaknesses of different repair techniques. Qi *et al.* [74] introduce the idea of a plausible versus correct patch, manually evaluating patches produced by previous heuristic techniques to highlight the risks that test cases pose when guiding repair search. Smith *et al.* [81], empirically and systematically study the overfitting issue for search-based repair techniques, including GenProg and RSRepair. In this dissertation, our study complements this previous study, in that we investigate the overfitting issue in the semantics-based repair family. [59] study the search space to find repair of the search-based approaches, showing that the search space is often large and correct repair sparsely occur within the search space. [49] empirically study the effectiveness of many synthesis engines when employed for semantics-based program repair, suggesting that many synthesis engines could be combined or used at the same time to enhance the ability of semantics-based repair to generate correct repairs. We leverage the technique from [49] to use multiple SyGuS engines in the context of a semantics-based repair approach.

Overfitting or manual annotation are not the only measure by which patch quality may be assessed. In proposing Angelix, [66] assess functionality deletion as a proxy for quality. [54] evaluate generated patches in a case study context, quantitatively assessing their impact in a closed-loop system for detection and repair of security vulnerabilities. [37] assess relative acceptability of patches generated by a novel technique via a human study. [26] conduct a human study of patch maintainability, finding that generated patches can often be as maintainable as human patches. While overfitting as measured by high-quality test suites provide one signal about patch quality, human acceptability and real-world impact are also important considerations, if not more so, and should also be considered in characterizing the pragmatic utility of novel APR techniques.

Chapter 3

History Driven Program Repair

One primary reason for overfitting is the reliance on test cases to guide the bug fixing process. In this work, we propose to use bug fixes in development history of many programs to guide and drive program repair, rather than relying on test cases alone. The main intuition is that recurring bug fixes are common in practice, and that previously-appearing fix patterns can provide useful guidance to an APR technique.

3.1 Introduction

Bugs are prevalent in software development. Mature commercial software systems regularly ship with both known and unknown defects [55], despite the support of multiple developers and testers typically dedicated to such projects [6]. To maintain software quality, bug fixing is thus inevitable and crucial. Yet, bug fixing is notoriously a difficult, time-consuming, and labor-intensive process, dominating developer time [91] and the cost of software maintenance. Many defects, including security-critical defects, remain unaddressed for extensive periods [31], and the resulting impact on the global economy is measured in the billions of dollars annually [84, 12]. There is a dire need to develop automated techniques to ease the difficulty and cost of bug fixing in practice.

To address the above-mentioned need, substantial recent work proposes techniques for Automated Program Repair (APR). These techniques seek to automatically fix bugs by producing source-level patches. For example, GenProg [52] uses a Genetic Programming [40] heuristic to conduct a search for a patch that causes the input program to pass all given test cases (including at least one that initially failed, exposing the defect to be addressed). Subsequently, Kim *et al.* extend the GP approach in Pattern-based Automatic program Repair (PAR), which uses bug fix templates manually learned from existing human-written patches [37] to guide the creation of the potential patches. These techniques are instructive examples of generate-and-validate and test-case-driven approaches to defect repair: They generate many candidate patches, and validate them against a set of test cases. The process is repeated many times, with a *fitness score* computed for each candidate patch based on the number of test cases that the associated modified program passes or fails. This score guides subsequent iterations, and thus the way the techniques traverse the search space of candidate repairs.

Despite the promise of existing APR techniques, current approaches are limited in several key ways [74]. To truly improve the quality of real-world software as well as the experience of modern software developers, an ideal technique must be both *effective* (i.e, able to fix many real bugs) as well as *efficient* (i.e., able to do so in a short amount of time). Even merely *plausible* patches—those that lead the buggy program to pass the provide test cases, but that are not necessarily globally correct as judged by an informed programmer—may take more than 10 hours to generate, and the resulting patches may still be incorrect [37, 74]. Although the risk of low quality patches can be mitigated by using more comprehensive test suites to guide the search process, even with full-coverage test suites, existing test-guided techniques may be susceptible to *overfitting* [81]. That is, produced patches may fail to functionally generalize beyond the test suite used to produce them. Although the current APR state-of-research is still in its infancy, it is important to work towards both effectiveness and efficiency to allow APR to be ultimately adopted.

In this chapter, we propose a novel technique for *history-based program repair*. Like several previous methods, our technique makes use of a stochastic search process to generate and then validate large numbers of patches, seeking one that causes previously-failing tests to pass. The most important feature differentiating our new technique from the previous work is that it evaluates the fitness or suitability of a candidate patch by assessing the degree to which it is similar to prior bug-fixing patches, taken from a large repository of real patches. This is in contrast with previous search-based approaches, which by and large use input test cases to assess intermediate patch suitability. Our intuition is that bug fixes are often similar in nature and past fixes can be a good guide for future fixes. This has at least partially informed a number of previous studies and approaches [10, 29, 37, 64]. The important novelty in our technique is that, instead of simply using previous fixes to inform the *construction* of candidate patches, we use fix history to help assess their potential *quality*, or fitness. We expect that the history-driven approach mitigates the risk of overfitting to the test suite, because it does not directly use the test suite score to guide individual selection for later iterations. This increases the probability that the resulting patches generalize to the desired program specification. Moreover, using the history to guide the repair search can also imbue the APR process with history-informed "common sense" to identify plausible but clearly—to humans—nonsensical patches.

To illustrate, consider the buggy code snippet in Figure 3.1, taken from Math version 85 in Defects4J benchmark [34]. This buggy snippet throws a `ConvergenceException` when one of the test cases is executed. One low-quality way to "fix" the problem that eliminates the symptom, and causes the test case to trivially pass, simply deletes the `throw` statement. However, this would be a nonsensical solution, and is not consistent with the patch the human developer committed for the same defect. Unfortunately, prior generate-and-validate and test-case-driven APR techniques cannot identify such a solution as nonsensical. In our history based approach, on the other hand, the fact that such edits very rarely appear in the historical bug fix data means that it receives a very low score in the search process. In this way, our technique is more likely to avoid plausible but nonsensical patches.

```
//Human fix: fa * fb > 0.0
if (fa * fb >= 0.0 ) {
   throw new ConvergenceException("...")
}
```

Figure 3.1: A bug in Math version 85

Our history-based APR technique works in three phases: (1) *bug fix history extraction* (2) *bug fix history mining* and (3) *bug fix generation*. The first two phases are conducted in advance of any particular bug-fixing effort. In the first phase, our technique mines historical bug fixes from revision control systems of hundreds of projects in GitHub. In the second phase, our technique identifies a clean set of data, seeking to find frequently appearing or common bug fixes, and infering a common representation to capture many similar such bug fixes. Bug fixes are represented as change graphs, which have the benefit of being generic and able to capture various kinds of changes along with their contexts. These change graphs, along with their frequencies, are used as a knowledge base for the third phase. In the third phase, our technique iteratively generates candidate patches, ranks them based on the frequency with which their constituent edits appear in the knowledge base inferred in the second phase, and returns a ranked list of plausible patches that pass all previously failed test cases as recommendations to developers.

We have evaluated our solution on 90 real bugs from 5 Java programs. We compare our technique against GenProg and PAR. GenProg is a popular generate-and-validate and test-case-driven APR technique that with a publicly available Java implementation.[1] Similarly, PAR is a generate-and-validate, test-case-driven technique developed for Java programs that explicitly makes use of edit templates manually synthesized from edit histories. Both are generic approaches that can, in theory, produce multi-line patches for bugs in programs. Our experiments show that our approach can correctly fix 23 bugs out of the 90 programs, while PAR and GenProg are only able to correctly fix 4 and 1 bugs, respectively. Moreover, our approach on average only needs 20 minutes to fix the 23 bugs. The results demonstrate the effectiveness and efficiency of our proposed approach.

The contributions of our work are as follows:

1. We propose a *generic* and *efficient history-based* automatic program repair technique that uses information stored in revision control systems of hundreds of software systems to generate plausible and correct patches. Our approach is generic since it can deal with bugs whose fixes involve multi-line changes. It is efficient since it can complete on average within less than 20 minutes.

2. We demonstrate that our approach is effective in fixing 23 bugs correctly, dramatically outperforming the performance of the baseline solutions.

3. Our approach supports Java instead of C. Java is the most popular programming language and its influence is growing over time.[2] Prior generate-and-validate and test-case-driven APR techniques mostly work on C programs with a few exceptions (e.g., PAR). Unfortunately, the implementation of PAR is not made publicly available. To facilitate reproducible research, we made the implementation of our approach available at https://github.com/xuanbachle/bugfixes

The structure of the remainder of this chapter is as follows. In Section 3.2, we elaborate the three steps of our proposed approach. In Section 3.3, we

[1]https://libraries.io/github/SpoonLabs/astor
[2]`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

present our experiment results which answer four research questions. We conclude in Section 3.4.

3.2 Proposed Approach

The overall goal of our approach is two-fold: to generate correct, high-quality bug fixes; and to quickly present such fixes to developers. To achieve this goal, we divide our framework into three main phases: (1) *bug fix history extraction*, (2) *bug fix history mining* and (3) *bug fix generation*. The first phase extracts a dataset of bug fixes made by human in the history from GitHub. This dataset is input to the second phase, which converts the bug fixes to a graph-based representation from which it automatically mines bug fix patterns. The mined bug fix patterns are input to the last phase.

In the last phase, we use a modified stochastic search technique [28] to evolve patches to a given buggy program, until we find a desired number of solutions. To reduce the risks of either overly constraining the search space or overfitting to the test suite, we use 12 existing mutation operators previously proposed in the mutation testing literature and used by prior repair techniques [69, 52, 37]. The *fitness* of the generated fix candidates is determined by the *frequency* with which the changes included in a given patch occur in the mined bug fix patterns produced by the second phase. Better fix candidates are thus those that frequently occur in the past fix patterns, and are thus more likely to be chosen to be validated against failed test cases, i.e., the test cases that reveal the bug in the original buggy program. Such selected patches are also more likely to be further developed and evolved in subsequent iterations. This fix candidate generation process is repeated until a desired number of candidates that pass all the failed test cases is identified. At the end of this phase, these candidates are presented to the developer as possible fixes for the bug, ranked by the frequency with which their edits appear in the bug fix history. The developer can then investigate the suggested fixes to find an actually correct fix.

In the next subsections, we describe each of the three phases of our framework in more detail (Sections 3.2.1–3.2.3); Section 3.2.4 describes our mutation operators.

3.2.1 Bug Fix History Extraction

In this phase, we collect human-made bug fixes from many open source projects on Github. The primary purpose of this phase is to collect and collate commits that are solely related to bug fix actions, excluding feature requests, refactoring, and other non-repair types of edits.

To collect bug fix history data from GitHub, we follow the procedure described by Ray *et al.* [76] to gather large, popular, and active open source Java projects. In particular, we use *Github Archive* [1], a database that frequently records public activities from GitHub, e.g., new commits, fork events, etc, to select only projects with the above characteristics. The popularity of a project is indicated by the number of *stars* associated with its repository, which corresponds to the number of GitHub users that have expressed interest in that project. In the interest of identifying only large, popular projects, we filter out those with fewer than five stars and exclude projects with repositories smaller than 100 Megabytes. Finally, we retain projects that are active as of September, 2014. This still leaves us with thousands of projects.

For each of the retained projects, we iterate through its source control history, seeking to collect commits that exclusively concern bug repair. This is a difficult problem in repository mining [11]. We therefore seek a complete set of bug-fixing commits using heuristics, though acknowledge that our approach is best-effort. We deem a commit a bug fix if it simultaneously satisfies the following conditions:

1. Its commit log contains the keywords such as *fix, bug fix*, while not con-

taining keywords such as *fix typo, fix build* or *non-fix.*

2. It includes the submission of at least one test case in that commit. Although the submitted test case does not necessarily mean the one that induces the bug, the inclusion of test case in the commit further increases the likelihood that the commit is a bug fix.

3. It involves changes on no more than two source code lines. The changed lines are counted, excluding code comments.

This last requirement warrants additional explanation. Commits that satisfy the first condition but involve many changed lines typically include changes beyond the bug fix, addressing feature addition, refactoring, etc [30, 35]. Thus, we filter out commits involving more than two changed lines. Ultimately, this leaves us with 3000 bug fixes across 700+ large, popular and active open source Java projects from GitHub.[3]

3.2.2 Bug Fix History Mining

In the second phase, we mine frequent bug fix patterns from the 3000 bug fixes, that appear in more than 700 projects, collected by previous phase. We first convert the collected bug fixes into a graph-based representation. We then apply an existing graph mining technique to the dataset to mine closed frequent patterns from the converted graphs.

Graph-based representation of bug fixes. Our goal in representing bug fixes is to succinctly abstract similar bug fixes into a common, abstract representation amenable to mining, which is especially challenging in the face of naming differences. Different bug fixes may vary in terms of the naming scheme in the underlying code, containing modifications to different variable names, method names, etc. For example, Figure 3.2 shows two bug fixes that both involve the change of method call parameter. Although there are differences in

[3]Dataset available: https://github.com/xuanbachle/bugfixes

the expressions (variables) that invoke the method calls, the method call names and parameter names, conceptually, these bug fixes can easily be classified as the same kind of bug fix, i.e., "method call parameter replacement."

Our first step in storing a bug fixing change is to capture its effects at the Abstract Syntax Tree (AST) level, which abstracts away many incidental syntactic differences (e.g., whitespace, bracket placement) that obscure a patch's semantic effect. To this end, we use GumTree,[4] an off-the-shelf, state-of-the-art tree differencing tool that computes AST-level program modifications [24]. GumTree represents differences between two ASTs via a series of actions including additions, deletions, updates or moves of individual tree nodes to transform one AST to another. To do this, given a bug fix, we first identify the file modified by the bug fix, and then retrieve the versions of the file before and after the modifications were made. Both versions of the modified file are then parsed to ASTs, denoted as the "buggy AST" and "fixed AST," respectively. We then use GumTree to compute the actions needed to transform the buggy AST into the fixed AST. For example, GumTree gives us the action needed to represent the *bug fix 1* in the Figure 3.2 as *update* from *x1* to *x2*.

However, this raw information provided by GumTree is insufficiently abstract on its own, since it is still specific to the variable names *x1* and *x2*. Additionally, the semantic context surrounding the action is unclear, that is, whether the action applies to a method call, an assignment, etc. To remedy this issue, we convert the series of actions produced by GumTree into a labelled directed graph that further abstracts over the edit actions, while being able to capture surrounding semantics. In this directed graph representation, an edge from a parent vertex to a child vertex is labelled by the kind of the action made to the child vertex. The context of the action is then captured by the parent vertex. To illustrate by example, Figure 3.3 depicts the graph that represents the *bug fix 1* in Figure 3.2. Similarly, this graph also represents the

[4]https://github.com/GumTreeDiff/gumtree

change made in *bug fix 2*. Thus, by using this graph representation, we can represent bug fixes in a common abstraction and capture contexts of the bug fixes. This graph-based representation will then help us in using graph mining techniques to mine frequent bug fix patterns.

```
//Bug fix 1: x1 replaced by x2, others remain the same
        - obj1.doX(x, x1)
        + obj1.doX(x, x2)

//Bug fix 2: y1 replaced by y2, others remain the same
        - obj2.doY(y, y1)
        + obj2.doY(y, y2)
```

Figure 3.2: Example of two bug fixes involving method call parameter replacement.

Mining closed frequent bug fix patterns. Given the full set of bug fixes, represented as graphs, we mine *closed frequent patterns* from the graphs. A pattern is *frequent* if it often occurs in the database; we heuristically set this count to at least two. A frequent pattern g is *closed* if there exists no proper supergraph of g that has the same number of supergraphs, i.e., *support*, as g. Thus, by definition, closed frequent patterns are the largest possible patterns that frequently occur in the database. In our domain, our goal is to mine the largest possible bug fix patterns to precisely capture behaviours of the changes. We therefore employ an extension of gSpan,[5] an implementation of a state-of-the-art frequent graph miner [97] for this task. We consider a pattern is frequent if it has support greater than or equal to two. We store information about patterns, including each pattern's vertices, edges, and supergraphs that contain the pattern. The number of supergraphs of a pattern constitutes the frequency of the pattern.

Algorithm 1: Bug Fix Generation. The `select` procedure returns one or more individuals from a population, either uniformly or weighted by a provided function. `applies` and `instant` are described in text. The tunable parameters are *PopSize* (population size), *M* (desired solutions), *E* (number of seeded candidates to the initial population), and *L* (number of locations considered in the mutation step).

```
Input  : BugProg: Buggy program
         FaultLocs: Fault locations
         NegTests: initially failing test cases
         FixPar: mined edit frequency distribution
         ops: possible operators
         params: Tunable parameters PopSize, M, E, L
Output: A ranked list of possible solutions
helper fun editFreq(cand)
    let N ← |cand|
    return $\frac{\sum_{i=0}^{N-1} FixPar(cand_i)}{N}$
helper fun mutate(cand)
    let locs ← select(FaultLocs, L)
    let pool ← ∅
    foreach $f \in$ locs do
        let $op_f \leftarrow \bigcup_{op \in \texttt{applies}(ops,loc)} \texttt{instant}(op, loc)$
        let cand' ← cand + select($op_f$, 1)
        pool ← pool ∪{ cand' }
    end
    return select(pool, 1, editFreq)
fun main
    let Solutions ← ∅
    let Pop ← {E empty patches}
    while / Pop / < PopSize do
        Pop ← Pop ∪ mutate([ ])
    end
    repeat
        foreach $c \in$ Pop do
            if $c \notin$ Solutions then
                if c passes NegTests then
                    Solutions ← Solutions ∪ c
                else
                    c ← mutate(c)
                end
            end
        end
    until /Solutions/ = M
    return Solutions
```

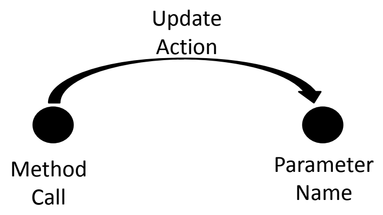


Figure 3.3: Graph-based representation of bug fixes in Figure 3.2

3.2.3 Bug Fix Generation

Overview. In this phase, we use a stochastic search approach loosely inspired by genetic programming [40] to evolve a patch for a given buggy program. The search objective is a patch that, when applied to the input program, addresses the defect, as identified by a set of failing test cases. GP is the application of genetic algorithms (GA) to problems involving tree-based solutions (programs, typically; in our application, these are small edit programs applied to the original buggy program). A GA is a population-based, iterative stochastic search method inspired by biological evolution. Given a tree-based *representation* of candidate solutions, GP uses computational analogues of biological mutation and crossover to generate new candidate solutions, and evaluates solutions using a domain-specific objective, or *fitness* function. Potential solutions with high fitness scores are more likely to be randomly retained into future iterations both alone, modified slightly (via *mutation*), or, in some applications, in combination with other solutions (via *crossover*).

In our approach, we represent a single candidate solution as a patch consisting of a sequence of edits to be made to the buggy program; this representation has been shown both efficient and effective in search-based program improvement [52]. Given a population of candidate solutions, we then use a selection process to create new candidates through mutation, and then to select mutated candidates to subsequent generations for additional evolution. The selection phase applies to the mutation step, in which a new edit is pseudo-randomly

[5]https://www.cs.ucsb.edu/ xyan/software/gSpan.htm

constructed and then added to an existing (possibly-empty) candidate patch. This selection is informed by the bug fix history database constructed as discussed in Section 3.2.2. Note that our algorithm does not perform crossover, using only mutation to create new individuals; we leave the development of a suitable crossover operator in our context to future work.

The details of this phase are further described in Algorithm 1. The primary inputs to the algorithm are the buggy program, where the bug is indicated by one or more failing test cases; a set of faulty locations, weighted by a preexisting fault localization procedure; a distribution of edit frequencies mined as discussed in the previous section, and a set of possible mutation operators. We presently assume that the faulty methods are known in advance, as file- and method-level localization represent an orthogonal problem; we then compute the faulty lines in each prospective faulty method using existing statistical fault localization techniques [3]. The stochastic algorithm includes several tunable parameters, described in context.

Given those inputs, the algorithm works in multiple iterations. The first iteration constructs an initial generation of *PopSize* candidate solutions by repeatedly constructing single-edit patches for the program (lines 16–18). Subsequent generations are created by adding new mutations to retained solutions in the current population. We describe mutation as it is used to create the initial population of single-edit patches; its application in subsequent iterations follows naturally.

Mutation. The mutation procedure adds an edit to a (possibly-empty) candidate patch to create a new patch candidate. It is described from line 4 to line 12 in Algorithm 1. At a high level, the mutation step creates a large number of candidate edits, from which a single edit is ultimately propagated into the candidate patch. First, the algorithm randomly selects a subset of L fault locations to which mutations may be applied (line 5), weighted by the score given by the statistical fault localization. We heuristically set L to 10 in our experiments, leaving a full parameter sweep to future work. Next, lines 7 to 10 generates a set of possible edits to select in this mutation step. This involves first identifying which mutation operators can be applied to each of the prospective faulty locations. For example, we should not use *append* to add any statements after a *return* statement, because doing so results in a dead code. This check is performed by the function `applies` on line 8. We reuse existing mutation operators (see Section 3.2.4 for a complete list, and details about their application), that have been proposed in prior mutation testing and program repair studies, to provide a diverse set of bug fix edit candidates. There are often several ways to instantiate a given operator. For example, *append* requires the selection of fix code to append at a given location. The `instant` function returns all possible instantiates of a given operator to the provided location, also on line 8. We select one of these edits (line 9) and create a new candidate by adding it to the current candidate.

This results in an intermediate *pool* of new pseudo-random patch candidates (initialized on line 6, updated on line 10) from which a single candidate will be retained. This retained candidate is thus the single result of the mutation step; it is the result of adding a new random edit to the (possibly-empty) candidate patch under mutation. To pseudo-randomly select an edit from this pool, we weight each edit by the *frequency* with which it appears in the mined bug fix patterns. This computation is performed in helper function `editFreq`, used in selection on line 12. Note that since exact graph matching (isomorphism) is notoriously difficult and expensive [86], we relax the conditions of matching fix candidates against past fix patterns. We instead say a fix candidate matches a fix pattern (graph) if the graph representing the candidate has more than half of its labels of vertices and edges matched with the fix pattern's vertices and edges respectively.

The frequency formula at line 3 works as follows: Given a fix candidate consisting of N edit operations, each edit operation contributes equally to the candidate's frequency. That is, we break down the block of N edits into each constitute edit and then fuse the frequency of each small edit together. The intuition is that, due to the randomness of the mutation procedure, generated fix candidates may contain bug-fix irrelevant edit operations, e.g., field or variable declarations. Ideally, these irrelevant edits should not affect the score of fix candidates containing them, since such edits contribute nothing or very little to the fixing effort. If we count the frequency of the fix candidate consisting of these edit operations by the whole block of N combined edits, it would make the fix candidate very rare when comparing the candidates against the historical bug fix patterns, and reduce the likelihood that the fix candidate will persist for future evolution. Our use of mean edit frequency mitigates the effect of adding bug-fix irrelevant edit operations with respect to the viability of the overall patch.

At line 12, we pseudo-randomly select one edit from the pool to add to the current candidate solution. This selection is informed by the computed frequency of a candidate patch that includes each edit in turn (the higher the frequency score of the overall patch that includes it, the more likely it is that the potential edit is selected from the pool). We use stochastic universal sampling [9] for this task. This selected candidate is thus the return value of the mutation procedure.

Main algorithm. Mutated candidates are created and processed by the main algorithm, described from line 13 to line 30. Line 15 adds E number of empty candidate patches to the initial population as seeds. We heuristically set E to 3 in our algorithm. Lines 16–18 create an initial population with $PopSize$ candidate patches by repeatedly mutating the empty patch. We heuristically set $PopSize$ to 40 in our algorithm. Next, from line 20 to line 28, we validate each candidate in the current population against the failed test cases. If a candidate passes all the failed test cases, we add it to the set of possible solutions (line 23). Otherwise, we mutate the candidate and carry the mutated

Table 3.1: 12 mutation operators employed in our framework

Operator Action	**Description**
	GenProg Mutation Operators
Insert statement	Insert a statement before or after a buggy statement
Replace statement	Replace a statement with a buggy statement
Delete statement	Delete a buggy statement
	Mutation Testing Operators
Insert Type Cast	Cast an object to a compatible type
Delete Type Cast	Delete type cast used on an object
Change Type Cast	Change type cast to another compatible type
Change Infix Expression	Change primitive operator (arithmetic, relational, conditional, etc) in an infix expression
Boolean Negation	Negate a boolean expression.
	PAR Mutation Operators
Replace Method Call parameter	Replace a parameter in a method call by another parameter with compatible types.
Replace Method Call Name/Invoker	Replace the name of a method call, or a method-invoking expression, by another method name or expression with compatible types.
Remove Condition	Remove a boolean condition in an existing *if* condition
Add Condition	Add a boolean condition to an existing *if* condition

candidate over the next iteration (line 25).

The process continues until a given number of fix candidates that pass all the previously failed test cases is reached. This is indicated at line 29, where the solutions' size reaches M desired solutions. We heuristically set M to 10 in our algorithm. Ultimately, these candidates are presented to the developer as possible fixes to the buggy program, ranked by the frequency of the underlying edits. The developer is then responsible for assessing the correctness of the suggested fixes. For example, the developer can pick any of the fixes appearing on top of the recommendation to validate the fixes by running them against previously passed test cases, and see if the fixes are actually semantically correct or not.

3.2.4 Mutation Operators

In this section, we describe the 12 mutation operators we employ to generate fix candidates in our framework; these operators are listed in Table 3.1. These operators have been used previously in mutation testing and well-known repair techniques; we use them to simultaneously provide a broad array of potential edit types, while mitigating the risk of overfitting the operators used in our experiments to the underlying dataset.

GenProg Mutation Operators. We employ the three mutation operators from GenProg [52]. The *delete* operator deletes a given potentially-buggy statement. The *insert* and *replace* operators work under the assumption that the repair is a piece of code that can be found from somewhere else in the same program. The insert operator inserts a randomly-selected statement before or after a given buggy statement.The *replace* operator replaces a potentially-buggy statement with another randomly-selected statement. For insert and replace, the original GenProg algorithm randomly chooses a statement from elsewhere in the same program, given certain semantic constraints (e.g., variable scoping). However, given a time limit, a large program can enormously reduce the possibility of selecting the correct statement.

We mitigate this problem in several ways. First, we reduce the scope of source statement selection to the same file with the target buggy statement. Previous studies have shown that this is adequate for many automated program repair problems [10]. Second, we heuristically prioritize in-scope statements. We view the problem of finding the source statement as two stages: First, we find the *clones* of the piece of code (method) surrounding the target buggy statement. Second, each of statements in the clones that have higher similarity is given higher probability to be a source statement. For statements that are not in any clones, we give them a default probability which is less than the probabilities of any statements found in clones. To find clones, we employ tree-based clone detection technique described by Jiang et al. in [33].

Mutation Testing Operators. We employ five mutation operators proposed in mutation testing research [69, 70]. The first three concern type casting: *delete type cast*, *insert type cast*, and *change type cast*. The latter two focus on inserting or changing casts only to compatible types. The *change infix expression operator* changes the operator used in a given infix expression. For example, an infix expression like $a \geq b$ involves an arithmetic operator that can be randomly changed, such as to $a > b$, $a < b$ or $a \leq b$. An infix expression $a \neq b$ that involves relational operator can be changed to $a == b$. An infix expression a && b that involves conditional operator can be changed to $a \mid\mid b$ and vise versa. The *boolean negation* operator tries to negate a boolean expression. For example, $true$ can be negated to $false$, and $isNegative(a)$ can be negated to $!isNegative(a)$.

PAR Mutation Operators. We employ four out of ten mutation operators proposed by Kim et. al. [37], leaving the employment of the remaining six operators as future work. These operators are applied to either method call or *if* condition. The first operator replaces a method call parameter, while the second operator replaces method call name, or the expression that invokes the method call. The last two operators deal with condition expression of `if` statement. An `if` condition expression containing more than two conditions can apply the *remove condition expression.* For example, `if(a || b){...}` can be changed to `if(a){...}` by removing condition b, which is randomly chosen from the condition. The *add condition expression* tries to add a condition to an `if` condition. The condition to be added is chosen from a pool of conditions collected from the same file with the faulty `if` statement. However, this pool of collected conditions can be inappropriate to fix a given bug, since they may reference out of scope variables.

To address this, our framework further cultivates the search space by inventing new conditions that have not appeared elsewhere in the same file. The idea is that the missing condition may very likely involve one of the variables used in the current `if` condition. Toward this end, we collect all variables used in the `if` condition. We then collect all boolean usages that involve the types of the collected variables from the same file. We then apply the usages with the collected variable names, and add these usages to the pool possible conditions that can be added to the current `if` condition.

3.3 Experiments & Analysis

In this section, we first describe our dataset (Section 3.3.1), followed by our experimental settings (Section 3.3.2), research questions (Section 3.3.3), and results (Section 3.3.4).

3.3.1 Dataset

We apply our approach to repair a subset of bugs from Defects4J [34], a large collection of defects in Java program intended to support research in fault localization and software quality. Defects4J has also been used in previous study of several automated program repair (APR) tools [61]. The dataset contains 357 real and reproducible bugs from 5 real-world open source Java programs. In our experiments, we use 90 bugs from Defects4J.[6] Table 3.2 depicts the number of bugs from each program in Defects4J and the number of bugs from each program that are used in our experiments. We use only these 90 bugs out of 357 bugs in Defects4J since we filtered out bugs that are too difficult for current state-of-the-art repair techniques to fix. That is, we first filter out bugs that involve more than six changed lines since they are typically too difficult for current automated program repair techniques to fix [74]. Second, we also filter out too difficult bugs considering the semantics of the bugs, even though they involve changes that are syntactically fewer than six lines. For example, one kind of difficult bugs could be adding a field in a class and use that field for fixing bugs in methods. We hypothesize that an effective and usable APR technique should be able to fix classes of bugs that are easier to fix first before it can handle very difficult bugs. We thus prefer this dataset, filtered according to rules suggested in previous empirical studies to a completely manually-constructed dataset to mitigate to some degree the threat over overfitting our technique to the bugs under repair [67]. We use the fix template database constructed as described in Sections 3.2.1–3.2.2.

[6]The bugs are made available here: https://github.com/xuanbachle/bugfixes

Table 3.2: Dataset Description. "#Bugs" denotes the total number of bugs in the Defects4J dataset. "#Bugs Exp" denotes the number of filtered bugs we used in our experiments.

Program	**#Bugs**	**#Bugs Exp**
JFreeChart	26	5
Closure Compiler	133	29
Commons Math	106	36
Joda-Time	27	2
Commons Lang	65	18
Total	357	90

3.3.2 Experiment Settings

We compare our approach against PAR [37] and GenProg [52]. Since PAR is not publicly available, we re-implemented a prototype of PAR for this experiment based on our framework. We also note that the original version of GenProg works on C programs and thus we used a publicly available implementation of GenProg[7] that works on Java program provided by Monperrus *et al.* [61].

We assign one trial for each approach to run on each bug. Specifically, each trial of our approach is assigned one 2.4 GHz Intel Core i5-2435M CPU and 8GBs of memory. Each trial is terminated either after 90 minutes or 10 generations or if 10 possible solutions were found. The size for each population is set to 40 for consistency with previous work [52, 37]. Since we consider current automated program repair techniques as only recommendation systems (since they cannot fix most of the bugs yet), an automated program repair technique needs to be efficient enough (c.f., [48]). We thus set the timeout for our experiment as 90 minutes for each trial. We note that since our approach, PAR and GenProg are all stochastic, multiple trials are needed to properly assess their performances [7].

[7]https://libraries.io/github/SpoonLabs/astor

3.3.3 Research Questions

In our experiments, we seek to answer the following research questions:

RQ1 How many bugs can our technique fix, correctly, as compared to the baselines?

We compare the effectiveness of our approach against PAR and GenProg in terms of number of bugs that each approach can correctly fix. To do this, we manually inspected generated patches to verify their correctness with respect to the corresponding bugs. A patch is deemed a correct fix if it satisfies the following conditions: (1) It results in a program that passes all test cases (both passing and initially failing). (2) it follows the behavior of the corresponding human-made fix. Checking the first condition is not difficult. However, the second condition involves an intrinsic qualitative judgement and a deep understanding of the program in question. Thus, for the second condition, we only consider fixes that are as close as possible to the human-made fixes. We leave a comprehensive human study on bug fixes quality to future work.

RQ2 Which bugs can the approaches fix in common? Which bugs can only be repaired by one of the approaches?

To gain insight into the process and limitations of the different approaches, we identify the defects for which our approach, PAR and GenProg all generate correct fixes. We describe case studies that illustrate potential reasons why some bugs can be fixed by one approach but not others.

RQ3 How long does it take to produce correct fixes?

In this research question, we investigate the average amount of time for each approach to run on the bugs that they can correctly fix. An approach is deemed efficient if it needs a reasonable computation time to find correct fixes. We consider current automated program repair techniques as recommendation systems, and a recommender that takes several hours to produce recommendations is ineffective.

RQ4 What are the rankings of the correct fixes among the solutions that our approach presents to the developer?

Our approach generates a ranked list of possible solutions to a given bug. The higher a correct fix is ranked, the better, requiring less effort from the developer to try the solutions one by one from the top to the bottom. Thus, in this research question, we investigate the ranking of the correct fixes among the possible solutions that our approach presents to the developer.

We report on two types of ranking. First, we present the ranking in the order that fixes are generated temporally. If effective, this ranking is helpful in case the developer is rushing to clear the bug, since he or she can just try whatever suggestions appear earlier instead of waiting for the whole process to complete. Second, we assess a ranking based on the frequency with which fix edits appear in the historical data.

3.3.4 Results

RQ1: Number of Bugs Correctly Fixed. Table 3.3 depicts the number of bugs for which each approach can generate correct fixes. In total, out of the 90 bugs, our approach generates correct fixes for 23 bugs; PAR can only correctly fix 4 bugs; GenProg generates only one correct fix. For the 23 bugs fixed by our approach, 11 (out of 12) mutation operators help fix these bugs. Each of these 11 operators helps fix no more than 5 bugs in the 23 bugs. Thus, it is not the case that the use of only a few operators can help fix all of the 23 bugs fixed by our approach. This supports our belief that while our approach is more effective than the baselines, its effectiveness is less likely to be biased by the experimental dataset.

Although the results for the previous techniques are somewhat worse than expected, we note that our timeout is set at 90 minutes and that we run only one trial on each bug. In previous experiments, GenProg and PAR set time outs at 12 hours, and run 10 trials in parallel for each bug [52, 37]. We can expect greater success if we increase the number of trials. However, our results are

Table 3.3: Effectiveness of our approach, PAR and GenProg in terms of number of defects repaired from each program. Our approach (named HDRepair) generates correct patches for 23 out of 90 bugs. Overall, 13 out of the 23 bugs fixed by HDRepair have correct patches ranked number 1, correct patches for the other 10 are ranked from 3 to 7 out of 10 solutions output by HDRepair per bug.

Program	**Our Approach**	**PAR**	**GenProg**
JFreeChart	2/5	-/5	-/5
Closure Compiler	7/29	1/29	-/29
Commons Math	6/36	2/36	-/36
Joda-Time	1/2	-/2	-/2
Commons Lang	7/18	1/18	1/18
Total	23/90	4/90	1/90

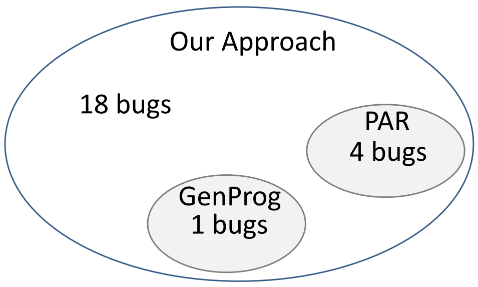


Figure 3.4: Common Bugs Fixed by Program Repair Techniques

consistent with a recent study, demonstrating that GenProg produces correct patches for 4 of 357 bugs in Defects4J with a 2 hour timeout and single trial per bug [61]. Although our results for GenProg can be substituted by that of [61], we reproduced GenProg experiment in our study to assure that all repair tools are given the same resources, e.g., computer RAM and CPU. In sum, the results show that our approach substantially outperforms PAR and GenProg in terms of number of bugs correctly fixed.

RQ2: Case Studies. Figure 3.4 shows that the bugs that PAR and GenProg correctly fix are a subset of those that our approach correctly fixes. There are 18 bugs that our approach can fix that PAR and GenProg do not. We present observations on this in the form of illustrative case studies.

Lack of Mutation Operators. In many cases, PAR's mutation operators/templates are inadequate for fixing these bugs in the same way that the developer did. For example, consider the human-produced fix for Commons Math Version 5:

```
if(real == 0.0 && imaginary == 0.0){
-  return NaN;
+  return INF;
}
```

Here, the human replaced one `return` statement with another. PAR has no mutation operator for this, while our approach has the *replace statement* operator adopted from GenProg, which helped generate this fix. Note that GenProg timed out on this bug, and thus did not fix the bug in our experiments.

Timeout. Even when the previous techniques possess the necessary mutation operators to potentially fix the bugs in the same way the developers did, in several cases they timed out before finding the fixes. For example, consider the developer-produced fix for Closure Compiler version 14:

```
for(Node finallyNode :  cfa.finallyMap.get(parent)){
-     cfa.createEdge(fromNode, Branch.UNCOND, finallyNode)
+     cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
}
```

The developer replaced the method call parameter `Branch.UNCOND` with another parameter, `Branch.ON_EX`. PAR includes potentially appropriate templates, such as change method call name or replace parameter for method call. There are thus many possibilities for PAR to generate fix candidates for this buggy statement. However, even if PAR can generate the correct fix candidate among the pool of possible fix candidates, the correct fix candidate was not evaluated, as PAR timed out while evaluating other, incorrect candidates. We leave a more extensive study with longer timeouts and more random trials to future work.

Plausible vs Correct Fixes. Automated program repair techniques can generate both plausible and correct patches. A plausible patch leads the patched program to pass all test cases, but does not necessarily correspond to a true fix, consistent with the underlying specification and developer intent. A correct fix, on the other hand, is the one that correctly fixes the semantics of the buggy program. For example, consider the following code, including a plausible patch generated by GenProg for Math version 85:

```
//Fix by human and our approach: change condition to fa * fb > 0.0
   if (fa * fb >= 0.0) {
   //Plausible fix by GenProg
-   throw new ConvergenceException("...")
   }
```

GenProg's plausible patch simple deletes the `throw` statement. This fix makes the program pass all the given test cases, at least in part because the test cases do not truly check the underlying behavior. However, as compared to the human fix for the same bug, this fix is unlikely to correspond to developer intent or the underlying program specification. Additionally, the deletion of `throw` statements rarely happens in historical practice. A more correct fix for this bug changes the arithmetic operator so that the exception is thrown in a correct manner that indeed satisfies the desired behaviour of the program; this is shown in the comment in the snippet, above the if condition.

In our approach, the *delete statement* mutation operator adopted from GenProg and the *change infix (arithmetic) expression* operator adopted from mutation testing both lead to the generation of a plausible patch: one similar to GenProg's, and the other similar to the human fix. However, partially due to the guidance provided by historical bug fixes, we avoid the plausible but incorrect patch and correctly choose the correct patch since the historical bug fix patterns suggest that changing an arithmetic happens more frequently in bug fixing practice. We also note that PAR does not generate any patch for this bug. Although PAR has the *expression replacer* operator which replace an *if* condition with another condition collected from the same scope, this operator does not help PAR generate patches for this bug since there is no correct condition appearing elsewhere in the same scope (same file).

Unfixed bugs in common. We observe that a common reason for why our approach, PAR and GenProg cannot fix bugs is a lack of ingredients that help synthesize the fix. For example, consider the human repair for Closure Compiler version 42:

```
+  if(loopNode.isForEach()){
+     errorReporter.error("unsupported...", sourceName,
     loopNode.getLineno(),"", 0);
+     return newNode(...);
   }
```

The developer added an entire `if` statement to fix the bug. At first sight, the bug may be fixable by the program repair techniques in the same way as the developer did, if the same `if` statement appears elsewhere in the search space. However, it is indeed not the case. Thus, the three approaches failed to generate fixes for this bug.

RQ3: Average Amount of Time to Correct Fixes. In this research question, we report the average amount of time that our approach, GenProg, and PAR need in order to generate the correct fixes. GenProg requires less than 10 minutes to produce the fix for the one bug that it can correctly fix. PAR requires on average 10 minutes to generate correct fixes for the 4 bugs that it successfully fixes. Our approach needs on average 20 minutes to generate correct fixes for each of the 23 bugs.

This indicates that PAR and GenProg are still efficient and effective for a certain class of bugs. For example, bugs that have a small search space to be traversed to find correct fixes could be quickly fixed by PAR or GenProg. Our approach, on the other hand, is resilient to many classes of bugs with the help of both the mutation operators and the guidance of historical bug fix data. Note, however, that although our technique takes longer than the baselines, 20 minutes is still well within the range of a suitably efficient technique. Also, the average time is computed over the time needed by our approach to fix the more difficult bugs that cannot be fixed by PAR and GenProg even within 90 minutes (timeout cases). The key to good efficiency of our approach is that we generate a diverse set of possible fix candidates, and then use historical data to help pick the likely good fix candidates and test them against only the failed test cases, which originally make the buggy program fail. Thus, we do not waste too much time on evaluating nonsensical candidates. However, we do depend on the developer to assess the final patches for suitability with respect to the initially passing test cases.

RQ4: Rankings of Correct Fixes among Recommended Solutions. In this research question, we report the rankings of the correct fixes among the possible solutions that our approach presents to the developer. Recall that for each bug, we attempt to generate 10 possible solutions. We investigate two criteria for ranking possible solutions: *time* in which the fixes are produced, and *edit frequency* in the historical database.

Using time, the correct fixes are ranked number one for 13 out of the 23 bugs that we can produce correct fixes. We note that there are 6 bugs that we can only generate one solution for each bug and this solution is indeed the correct fix of the bug. For the remaining 10 bugs, each bug has correct fix ranked from 3 to 7 among the 10 possible solutions presented to the developer. Using frequency, there are 11 bugs that have correct fixes ranked number one. The remaining 12 bugs have correct fixes ranked from 2 to 10, among the 10 possible solutions presented to the developer. Overall, these results suggest that ranking the correct fixes among possible solutions by either time or frequency is acceptable.

To further assure the correctness of patches generated by *HDRepair*, we use an automated test case generation tool namely *DiffTGen* [94] to obtain more test cases in addition to the repair test suite. The generated test cases are then used to assess patch correctness: if a test case is found to reveal behavioral differences between the programs patched by *HDRepair* and by human (groundtruth), then the patch by *HDRepair* is identified as incorrect. We experimented with all *HDRepair*-generated patches that are ranked first (by time) among the 10 solutions for each bug. The results show that *DiffTGen* identifies three incorrect patches. Note that the correct patches generated by *HDRepair* for the three incorrect patches are ranked from 3 to 7 among the 10 possible solutions output by *HDRepair* per bug .

3.4 Conclusions

Existing automated program repair (APR) techniques often unsuccessfully return correct patches despite running for a long period of time (e.g., more than 10 hours). In this work, we propose a *generic* and *efficient* APR technique that leverages information from *historical bug fixes*. Our solution takes as input a large set of repositories of software projects to create a knowledge base which is then leveraged to generate a ranked list of plausible bug fix patches given a buggy program and a set of test cases. It works on three phases: bug fix history extraction, bug fix history mining, and bug fix generation. We have evaluated the effectiveness of our proposed approach on a dataset of 90 bugs from five Java programs, and compared its effectiveness against two other generic generate-and-validate and test-case-driven APR techniques that work on Java programs. Our experiment results highlight that our approach can fix 23 bugs correctly, which are many more than the bugs that can be fixed by GenProg and PAR. On average, our solution can fix the 23 bugs within 20 minutes. These highlight the superior performance of our proposed approach in terms of effectiveness and efficiency as compared to existing generic APR solutions that can fix multi-line bugs in Java programs.

In the future, we plan to improve the effectiveness and efficiency of our solution further. We plan to do so by designing better ways to traverse the search space of potential patches. We also plan to incorporate data from not only 3,000 bug fixes but even a larger number taken from even many more programs.

Chapter 4

Overfitting in Semantics-based Repair

Search-based APR has been shown to be subject to overfitting, at various degrees [81]. Unfortunately, there exists no comprehensive study in the literature on the overfitting issue in semantics-based APR. In this work, we address this gap by studying various semantics-based APR techniques, complementing previous studies of the overfitting problem in search-based APR. We perform our study using IntroClass and Codeflaws benchmarks, two datasets well-suited for assessing repair quality, to systematically characterize and understand the nature of overfitting in semantics-based APR. We find that similar to search-based APR, overfitting also occurs in semantics-based APR in various different ways.

4.1 Introduction

Automated program repair (APR) addresses an important challenge in software engineering. Its primary goal is to repair buggy software to reduce the human labor required to manually fix bugs [84]. Recent advances in APR have brought this once-futuristic idea closer to reality, repairing many real-world software bugs [66, 52, 60, 37, 96, 48, 47, 46]. Such techniques can be broadly classified into two families, semantics-based vs. search-based, differentiated by the underlying approach, and with commensurate strengths and weaknesses. Semantics-based APR typically leverages symbolic execution and test suites to extract semantic constraints, or *specifications*, for the behavior under repair. It then uses program synthesis to generate repairs that satisfy those extracted specifications. Early semantics-based APR techniques used template-based synthesis [38, 39]. Subsequent approaches use a customized component-based synthesis [68, 96], which has since been scaled to large systems [66]. By contrast, search-based APR generates populations of possible repair candidates by heuristically modifying program Abstract Syntax Trees (AST)s, often using optimization strategies like genetic programming or other heuristics to construct good patches [89, 90, 52, 50, 73, 60].

Both search-based and semantics-based APR techniques have been demonstrated to scale to real-world programs. However, the quality of patches generated by these is not always assured. Techniques in both families share a common underlying assumption that generated patches are considered correct if they lead the program under repair pass all provided test cases. This raises a pressing concern about true correctness: an automatically-generated repair may not generalize beyond the test cases used to construct it. That is, it may be *plausible* but not fully *correct* [74]. This problem has been described as *overfitting* [81] to the provided test suites. This is an especial concern given that test suites are known to be incomplete in practice [84]. As yet, there is no way to know a priori whether and to what degree a produced patch overfits. However, the degree to which a technique produces patches that overfit has been used post facto to characterize the limitations and tendencies of search-based techniques [81], and to experimentally compare the quality of patches produced by novel APR methods [36].

There is no reason to believe that semantics-based APR is immune to this problem. Semantics-based approaches extract behavioral specifications from the same partial test suites that guide search-based approaches, and thus the resulting specifications that guide repair synthesis are themselves also partial. However, although recent work has assessed proxy measures of patch quality (like functionality deletion) [66], to the best of our knowledge, there exists no comprehensive, empirical characterization of the overfitting problem for semantics-based APR in the literature.

We address this gap. In this article, we comprehensively study overfitting in semantics-based APR. We perform our study on Angelix, a recent state-of-the-art semantics-based APR tool [66], as well as a number of syntax-guided synthesis techniques used for program repair [49]. We evaluate the techniques on a subset of the IntroClass [53] and Codeflaws benchmarks [82], two datasets well-suited for assessing repair quality in APR research. Both consist of many small defective programs, each of which is associated with two independent test suites. The multiple test suites renders these benchmarks uniquely beneficial for assessing patch overfitting in APR. One test suite can be used to guide the repair, and the other is used to assess the degree to which the produced repair generalizes. This allows for controlled experimentation relating various test suite and program properties to repairability and generated patch question.

In particular, IntroClass consists of student-written submissions for introductory programming assignments in the C programming language. Each assignment is associated with two independent, high-quality test suites: a *black-box* test suite generated by the course instructor, and a *white-box* test suite generated by automated test case generation tool KLEE [13] that achieves full branch coverage over a known-good solution. IntroClass has been previously used to characterize overfitting in search-based repair [81]. The Codeflaws benchmark consists of programs from the Codeforces[1] programming contest. Each program is also accompanied by two set of test suites: one for the programmers/users to validate their implementations, and the other for the con-

[1]`http://codeforces.com/`

test committee to validate the users' implementations.

Overall, we show that overfitting does indeed occur with semantics-based techniques. We characterize the relationship between various factors of interest, such as test suite coverage and provenance, and resulting patch quality. We observe certain relationships that appear consistent with results observed for search-based techniques, as well as results that stand counter to those achieved on them, e.g., using whitebox tests as training tests reduces the overfitting rate of semantics-based repair, while increases the overfitting rate of search-based repair.[2] These results complement the existing literature on overfitting in search-based APR, completing the picture on overfitting in APR in general. This is especially important to help future researchers of semantics-based APR to overcome the limitations of test suite guidance. We argue especially (with evidence) that semantics-based program repair should seek stronger or alternative program synthesis techniques to help mitigate overfitting.

Our contributions are as follows:

- We perform the first study on overfitting in semantics-based program repair. We show that semantics-based APR can generate high-quality repairs, but can also produce patches that overfit.
- We assess relationships between test suite size and provenance, number of failing tests, and semantics-specific tool settings and overfitting. We find, in some cases, results consistent with those found for search-based approaches. In other cases, we find results that are interestingly inconsistent.
- We substantiate that using multiple synthesis engines could be one possible approach to increase the effectiveness of semantics-based APR, e.g., generate correct patches for a larger number of bugs. This extends Early Results findings from [49].
- We present several examples of overfitting patches produced by semantics-

[2]Please refer to research questions 3 and 4 for more details.

based APR techniques, with implications and observations for how to improve them. For example, we observe that one possible source for overfitting in semantics-based APR could be due to the "conservativeness" of the underlying synthesis engine, that returns the first solution found (without consideration of alternatives).

The remainder of this chapter proceeds as follows. Section 4.2 describes background on semantics-based program repair. Section 4.3.1 explains the data we use in our experiments; the remainder of Section 4.3 presents experimental results, and insights behind them. We conclude and summarize in Section 4.4.

4.2 Semantics-based Program Repair

We focus on understanding and characterizing overfitting behavior in semantics-based automated program repair (APR). Semantics-based APR has recently been shown by [66] to scale to the large programs previously targeted by search-based APR techniques [52, 60]. This advance is instantiated in Angelix, the most recent, state-of-the-art semantics-based APR approach in the literature.

Angelix follows a now-standard model for test-case-driven APR, taking as input a program and a set of test cases, at least one of which is failing. The goal is to produce a small set of changes to the input program that corrects the failing test case while preserving the other correct behavior. At a high level, the technique identifies possibly-defective expressions, extracts value-based specifications of correct behavior for those expressions from test case executions, and uses those extracted specifications to synthesize new, ideally corrected expressions. More specifically, Angelix first uses existing fault localization approaches, like Ochiai [3] to identify likely-buggy expressions. It then uses a selective symbolic execution procedure in conjunction with provided test suites to infer correctness constraints, i.e., *specifications*.

We now provide detailed background on Angelix's mechanics. We first detail the two core components of Angelix: specification inference (Section 4.2.1) and program synthesis (Section 4.2.2). We explain various tunable options that Angelix provides to deal with different classes of bugs (Section 4.2.3). We then provide background on the variants of semantics-based APR we also investigate our experiments: SemFix (Section 4.2.4), and Syntax-Guided Synthesis (SyGuS) as applied to semantics-based APR (Section 4.2.5).

4.2.1 Specification Inference via Selective Symbolic Execution

Angelix relies on the fact that many defects can be repaired with only a few edits [63], and thus focuses on modifying a small number of likely-buggy expressions for any particular bug. Given a small number of potentially-buggy expressions identified by a fault localization procedure, Angelix performs a selective symbolic execution by installing symbolic variables α_i at each chosen expression i.[3] It concretely executes the program on a test case to the point that the symbolic variables begin to influence execution, and then switches to symbolic execution to collect constraints over α_i. The goal is to infer constraints that describe solutions for those expressions that could lead all test cases to pass.

These value-based specifications take the form of a *precondition* on the values of variables before a buggy expression is executed, and then a *postcondition* on the values of α_i. The precondition is extracted using forward analysis on the test inputs to the point of the chosen buggy expression; The postcondition is extracted via backward analysis from the desired test output by solving the model: $PC \wedge O_a == O_e$. PC denotes the path condition collected via symbolic execution, O_a denotes the actual execution output, and O_e denotes the expected output. The problem of program repair now reduces to a synthesis

[3]Angelix can target multiple expressions at once; we explain the process with respect to a single buggy expression for clarity, but the technique generalizes naturally.

problem: Given a precondition, Angelix seeks to synthesize an expression that satisfies the postcondition (described in Section 4.2.2)

Angelix infers specifications for a buggy location using a given number of test cases, and validates synthesized expressions with respect to the entire test suite. Angelix chooses the initial test set for the specification inference based on coverage, selecting tests that provide the highest coverage over the suspicious expressions under consideration. If any tests fail over the course of validation process, the failing test is incrementally added to the test set used to infer specifications for subsequent repair efforts, and the inference process moves to the next potentially-buggy location. This process is repeated until a repair that leads the program to pass all tests is found. We further discuss the number of tests used for specification inference in Section 4.2.3.

4.2.2 Repair Synthesis via Partial MaxSMT Solving

Angelix adapts component-based repair synthesis [32] to synthesize a repair conforming to the value-based specifications extracted by the specification inference step. It solves the synthesis constraints with Partial Maximum Satisfiability Modulo Theories (Partial MaxSMT) [65] to heuristically ensure that the generated repair is minimally different from the original program.

Component-based synthesis. The synthesis problem is to arrange and connect a given set of *components* into an expression that satisfies the provided constraints over inputs and outputs. We illustrate via example: Assume the available components are variables x and y, and binary operator "$-$" (subtraction). Further assume input constraints of $x == 1$ and $y == 2$, and an output constraint of $f(x, y) == 1$. $f(x, y)$ is the function over x and y to be synthesized. The component-based synthesis problem is to arrange x, y, and "$-$" (the components) such that the output constraint is satisfied with respect to the input constraints. For our example, one such solution for $f(x, y)$ is $y - x$;

Another is simply x, noting that the synthesized expression need not include all available components. The synthesis approach encodes the constraints and available components in such a way that, if available, a satisfying SMT model is trivially translatable into a synthesized expression, that synthesized expression is well-formed, and it provably satisfies the input-output constraints

Partial MaxSMT for minimal repair. Angelix seeks to produce repairs that are small with respect to the original buggy expressions. Finding a minimal repair can be cast as an optimization problem, which Angelix addresses by leveraging Partial MaxSMT [65]. Partial MaxSMT can solve a set of *hard* clauses, which *must* be satisfied, along with as many *soft* clauses as possible. In this domain, the hard clauses encode the input-output and program well-formedness constraints, and the soft clauses encode structural constraints that maximally preserve the structure of the original expressions. Consider the two possible solutions to our running example: $f(x, y) = y - x$, or $f(x, y) = x$. If the original buggy expression is $x - y$, synthesis using Partial MaxSMT might produce $f(x, y) = y - x$ as a desired solution, because it maximally preserves the structure of the original expression by maintaining the "$-$" operator.

4.2.3 Tunable Parameters in Angelix

We investigate several of Angelix's tunable parameters in our experiments. We describe defaults here, and relevant variances in Section 4.3.

Suspicious location group size. Angelix divides multiple suspicious locations into groups, each consisting of one or more locations. Angelix generates a repaired expression for each potentially-buggy expression in a group. During specification inference, Angelix installs symbolic variables for locations in each group, supporting inference and repair synthesis on multiple locations. Given a group size N, Angelix can generate repairs that touch no more than N locations. For example, if $N = 2$ (the default setting), Angelix can generate a repair that modifies either one or two expressions. Angelix groups buggy expressions by either suspiciousness score, or proximity/location. By default, Angelix groups by location.

Number of tests used for specification inference. The number of tests used to infer (value-based) specifications is important for performance and generated patch quality. Too many tests may overwhelm the inference and synthesis engines; too few may lead to the inference of weak or inadequate specifications expressed in terms of input-output examples, which may subsequently render the synthesis engine to generate poor solutions that do not generalize. As described above, Angelix increases the size of the test suite incrementally as needed. By default, two tests are used to start, at least one of which must be failing.

Synthesis level. The selection of which components to use as ingredient components for synthesis is critical. Too few components overconstrains the search and reduces Angelix's expressive power; too many can overwhelm the synthesis engine by producing an overly large search space. Angelix tackles this problem by defining *synthesis levels*, where each level includes a particular set of permitted ingredient components. For a given repair problem, the synthesis engine searches for solutions at each synthesis level, starting with the most restrictive and increasing the size of the search space with additional components until either a repair is found or the search is exhausted. By default, Angelix's synthesis levels include *alternatives, integer-constants,* and *boolean-constants* levels. The *alternatives* synthesis level allows Angelix's synthesis engine to use additional components similar to existing code, e.g., "$\leq$" is an alternative component for the component "$<$". The *integer-constants* and *boolean-constants* levels enable additional integer and boolean constants available to the synthesis engine, respectively.

Defect Classes. Angelix can handle four classes of bugs, related to, respectively, *assignments*, *if-conditions*, *loop-conditions*, and *guards*. The "assignments" defect class considers defective right-hand-sides in assignments. "if-conditions" and "loop-conditions" considers buggy expressions in conditional statements. The "guards" defect class considers the addition of synthesized guards around buggy statements. For example, Angelix might synthesize a guard *if($x > 0$)* to surround a buggy statement $x = y + 1$, producing *if($x > 0$)* $\{x = y + 1\}$. The more defect classes considered, the more complicated the search space, especially given the "guard" class (which can manipulate arbitrary statements). By default, Angelix considers assignments, if-conditions, and loop-conditions.

4.2.4 SemFix: Program Repair via Semantic Analysis

SemFix, a predecessor of Angelix, is a synergy of fault localization, symbolic execution, and program synthesis. The primary differences between SemFix and Angelix are: (1) SemFix's specification inference engine works on only a single buggy location (Angelix can operate over multiple buggy locations at once), (2) SemFix defines the specification as a disjunction of inferred path conditions. Angelix instead extracts sequences of angelic values that allow the set of tests to pass from each path, and uses them to construct a so-called "angelic forest." As a result, the size of Angelix specification is independent of the size of the program (depending only on the number of symbolic variables). This makes Angelix more scalable than SemFix, and (3) SemFix does not attempt to minimize the syntactic distance between a solution and the original buggy expression using Partial MaxSMT. These differences are particularly important for scalability (Angelix can repair bugs in larger programs than can SemFix), and patch quality, which this article explores in detail.

4.2.5 Syntax-Guided Synthesis for Semantics-Based Program Repair

Other synthesis approaches are also applicable to semantics-based program repair, with possible implications for repair performance [49]. We systematically evaluate these implications for repair quality, and thus now describe the Syntax-Guided Synthesis (SyGuS) [4] techniques we use in our experiments.

Given a specification of desired behavior, a SyGuS engine uses a restricted grammar to describe (and thus constrain) the syntactic space of possible implementations. Different SyGuS engines vary in the search strategies used to generate solutions that satisfy the specification and conform to the grammar. We investigate two such techniques:

- The *Enumerative* strategy [4] generates candidate expressions in increasing size, and leverages specifications and a Satisfiability Modulo Theory (SMT) solver to prune the search space of possible candidates. Since repeatedly querying an SMT solver regarding the validity of a solution with respect to a specification (the *validity query*) is expensive, it uses counter-examples to improve performance. That is, whenever a solution failed to meet the specification, a counter-example is generated and added to the next validity query.
- *CVC4* is the first SyGus synthesizer [78] implemented inside an SMT solver, via a slight modification of the solver's background theory. To synthesize an implementation that satisfies all possible inputs, it translates the challenging problem of solving universal quantifier over all inputs into showing the unsatisfiability of the negation of the given specification. It synthesizes a solution based on the unsatisfiability proof.

Recent SyGuS competitions suggest that the CVC4 and enumerative engines are the among the best, evaluated on SyGuS-specialized benchmarks.[4]

[4]http://www.sygus.org/

We follow the approach described in previous work [49] to integrate the *Enumerative* and *CVC4* synthesizers into Angelix. At a high level, Angelix infers value-based specifications as usual, and we automatically translate those specifications into a suitable SyGuS format, with optimizations to constrain the repair minimality. Different SyGuS engines can then be run on the same generated SyGuS script to synthesize a repair conforming to the inferred specifications, allowing for a controlled comparison of different synthesis approaches in the context of a semantics-based repair technique.

4.3 Empirical Evaluation

The primary purpose of our experiments is to systematically investigate and characterize overfitting in semantics-based APR. To this end, we use benchmarks that provide many buggy programs along with two independent test suites. For each run of each repair technique on a given buggy program, we use one set of provided test cases (the *training* tests) to generate a repair, and the other (the *held-out* tests) to assess the quality of the generated repair. If a repair does not pass the held-out tests, we say it is an *overfitting* repair that is not fully general; this is a proxy measure for repair quality (or lack thereof). Otherwise, we call it a *non-overfitting* or *general* repair.[5]

We describe our experimental dataset in Section 4.3.1. We then begin by assessing baseline patching and overfitting behavior generally (Section 4.3.2). We then evaluate relationships between overfitting and characteristics of input test suites and input programs (Section 4.3.3), as well as tunable tool parameters (Section 4.3.4). Finally, we present and discuss several informative test cases from the considered dataset (Section 4.3.5) and a qualitative case study on real-world bugs (Section 4.3.6).

[5]We use the words “repair” and “patch” interchangeably.

4.3.1 Experimental data

We obtained Angelix from `https://github.com/mechtaev/angelix/`, using the version evaluated in [66]. We set all tunable parameters to their defaults (Section 4.2.3) unless otherwise noted.

We conduct the majority of our experiments on buggy programs from a subset of the IntroClass benchmark [53], and the Codeflaws benchmark [82].[6] Both benchmarks consist of small programs, but are particularly suitable benchmark for assessing repair quality via overfitting, because they each provide two test suites for each buggy program. One set can be used to guide the repair, while the second set is used to assess the degree to which it generalizes.

IntroClass: IntroClass consists of several hundred buggy versions of six different programs, written by students as homework assignments in a freshmen programming class. Each assignment is associated with two independent high-coverage test suites: a black-box test suite written by the course instructor, and a white-box test suite generated by the automated test generation tool KLEE [13] on a reference solution. We filtered IntroClass to retain only textually unique programs. We then further filter to retain those programs with outputs of type boolean, integer, or character because Angelix's inference engine does not fully support output of other types such as String or float due to the limited capability of constraint solving technique used in Angelix's underlying symbolic execution engine. This leaves us with 315 program versions in the dataset, shown in column "Total" in Table 4.1(a) (grouped by assignment type).

Codeflaws: The Codeflaws benchmark contains 3,902 defects collected from the Codeforces programming contest,[7] categorized by bug types [82]. Since running all bugs is computationally expensive, we select for our experiments 665 bugs belonging to different bug categories. The selected bugs are from the "replace

[6]We discuss the real-world bugs we describe qualitatively in Section 4.3.6.
[7]http://codeforces.com/

```
// An example of ORRN defect type
-if(a > b)
+if(a >= b)

// An example of OLLN defect type
-if(a || b)
+if(a && b)

// An example of OILN defect type
-if(a)
+if(a || b)
```

Figure 4.1: Examples of defect types from the Codeflaws dataset used in our experiments.

relational operator" bug category (*ORRN*), and the "replace logical operator" (*OLLN*) and "tighten or loosen condition" (*OILN*) categories. Examples of the selected defect types are shown in Figure 4.1. These three selected defect categories are best suited to repair via semantics-based techniques (note that the majority of bugs fixed by Angelix in [66] belongs to the "if-condition" defect type).

Similar to IntroClass, each program in Codeflaws is accompanied by two test suites: one suite is available to the contest's users to assess their implementation, and the other is *only* available to the contest's committee to assess the implementations submitted by users.

4.3.2 Baseline patching and overfitting

Our first three research questions (1) establish baseline patch generation results, (2) evaluate whether there exists an apparent relationship between the number of tests a program fails and repair success, and (3) assess the degree to which the semantics-based techniques under consideration produce patches that overfit to a training test suite.

Research Question 1: How often do Angelix (including various synthesis engines) and SemFix generate patches that lead the buggy programs to pass all training test cases?

IntroClass: In these initial experiments, we use the black-box tests as training tests, and the white-box tests as held-out tests. Of the 315 program versions, 225 programs that have at least one failing black-box test case. The center portion of Table 4.1(a) shows results (we discuss white-box results in Research Question 6), in terms of the number of patches generated by Angelix, CVC4, Enumerative, and SemFix using black-box tests for training. In total, Angelix generates patches for 81 out of 225 versions (36%). Note that Angelix generated no patches for the *syllables* and *checksum* programs; our manual investigation suggests that this is primarily due to imprecision in the built-in fault localization module. Beyond this, success rate varies by assignment type. Angelix has the most patch generation success (70.4%) for programs written for the *median* problem. The overall results indicate that Angelix generates patches frequently enough for us to proceed to subsequent research questions.

Angelix incorporating the CVC4 and Enumerative SyGuS engines generated patches for 71 and 59 versions, respectively, a lower patch generation suc-

Table 4.1: Baseline repair results on IntroClass and Codeflaws.

(a) Baseline repair results on IntroClass. Total benchmark program versions considered (Total), baseline repair results for programs that fail at least one black-box test (Black-box, center columns), and those that fail at least one white-box test (White-box, last columns). The sets of programs that fail at least one test from each set are not disjoint.

		Black-box bugs					White-box bugs				
Subject	Total	#	Angelix	CVC4	Enum	SemFix	#	Angelix	CVC4	Enum	SemFix
smallest	67	56	37	39	29	45	41	37	37	36	37
median	61	54	38	28	27	44	45	35	36	23	38
digits	108	57	6	4	3	10	90	5	2	2	8
syllables	48	39	0	0	0	0	42	0	0	0	0
checksum	31	19	0	0	0	0	31	0	0	0	0
total	315	225	81	71	59	99	249	77	75	61	83

(b) Baseline repair results on Codeflaws. The tests available to the users serve as training tests; the contest committee tests serve as held-out tests.

Subject	Total	Angelix	CVC4	Enum	SemFix
CodeFlaws	651	81	91	92	56

Table 4.2: Baseline overfitting results on IntroClass (top) and Codeflaws (bottom). In both tables, A / B denotes A overfitting patches out of B total patches generated.

(a) IntroClass overfitting rates for each APR approach, using black box (center columns) and white box (right-most columns) as training tests. We omit syllables and checksum, for which no patches were generated.

	Black box				White box			
Subject	Angelix	CVC4	Enum	SemFix	Angelix	CVC4	Enum	SemFix
smallest	27 / 37	33 / 39	24 / 29	36 / 45	31 / 37	33 / 37	33 / 36	33 / 37
median	29 / 38	21/ 28	21 / 27	40 / 44	25 / 35	36 / 36	23 / 23	38 / 38
digits	5 / 6	3 / 4	3 / 3	10 / 10	0 / 5	2 / 2	2 / 2	2 / 8
Overfitting	75%	80%	81%	90%	72.7%	95%	93.5%	87%

(b) Codeflaws overfitting rates for each APR approach.

	Angelix	CVC4	Enum	SemFix
	44 / 81	76 / 91	80 / 92	38 / 56
Overfitting	54%	83.5%	87%	68%

cess rate comparatively (31.6% for CVC4, and 26.2% for Enumerative). SemFix, on the other hand, generates patches for 99 versions with slightly higher patch generation rate (44%). Despite the lower patch generation rates, CVC4 and Enumerative do generate patches for programs for which Angelix cannot. This raises an interesting question regarding whether it might it be beneficial to use multiple synthesis techniques to increase the effectiveness of semantics-based APR. In subsequent research questions, we investigate whether Angelix, CVC4, Enumerative, and SemFix do indeed generate non-overfitting patches for distinct program versions.

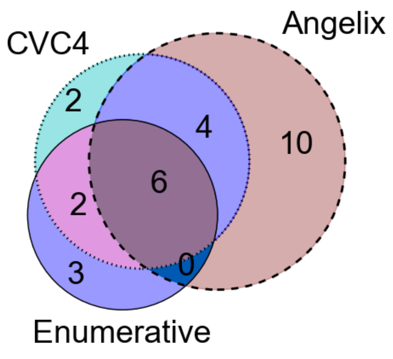
Codeflaws: For Codeflaws, we use the tests available to users as training tests, and the tests that are only available to the contest's committee as held-out tests. Table 4.1(b) shows results. Angelix, CVC4, Enumerative and SemFix succeed in generating patches for 12.5%, 14%, 14%, and 9% of the buggy programs, respectively. Although this is a much lower patch generation rate as compared to the IntroClass results, the number of generated patches is adequate to allow us to proceed to subsequent research questions.

Research Question 2: How often do the produced patches overfit to the training tests, when evaluated against the held-out tests?

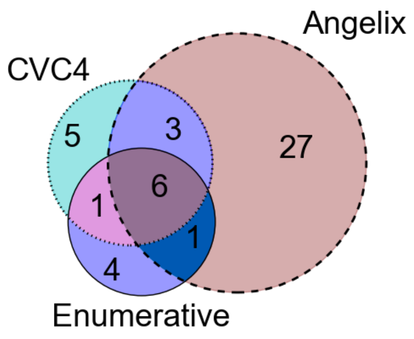
In this question, we evaluate whether the generated patches generalize, indicating that they are more likely to be correct with respect to the program specification. An ideal program repair technique would often generate general patches, and produce overfitting patches infrequently. We test all patches produced for Research Question 1 against the held-out to measure rate.

Results. Table 4.2(a) and Table 4.2(b) show the number of patches produced for each subject program that fail at least one held-out test for the IntroClass and Codeflaws datasets, respectively. On IntroClass, 61 of the 81 (75%) Angelix-produced patches overfit to the training tests, while 80%, 81%, and 90% of the CVC4-, Enumerative-, and SemFix-produced patches do, respectively. On Codeflaws, 44 of the 81 (54%) Angelix-produced patches overfit, while 83.5%, 87%, and 68% of patches generated by CVC4, Enumerative, and SemFix do, respectively. This suggests that, although semantics-based repair has been shown to produce high-quality repairs on a number of subjects, overfitting to the training tests is still a concern. We present case studies to help characterize the nature of overfitting in semantics-based APR in Section 4.3.5.

One possible reason that CVC4 and Enumerative underperform Angelix's default synthesis engine is that the SyGuS techniques do not take into account the original buggy expressions. We observed that the resulting patches can be very different from the originals they replace, which can impact performance arbitrarily. However, the CVC4 and Enumerative techniques do generate non-overfitting patches for programs that default Angelix cannot produce non-overfitting patches, as shown in Figure 4.2(a) and Figure 4.2(b). Similarly, SemFix, CVC4, and Enumerative also have non-overlapping non-overfitting patches (results omitted). This phenomenon also happens between Angelix and SemFix. This suggests that using multiple synthesis engines to complement one another may increase the effectiveness of semantics-based APR.



(a) Non-overfitting patches generated by Angelix, CVC4, and Enumerative on IntroClass.



(b) Non-overfitting patches generated by Angelix, CVC4, and Enumerative on Codeflaws.

Figure 4.2: Non-overfitting patches by Angelix, CVC4, and Enumerative on IntroClass and Codeflaws benchmarks.

4.3.3 Training test suite features

Our next three research questions look at the relationship between features of the training test suite and produced patch quality, looking specifically at (4) test suite size, (5) number of failing tests, and (6) test suite provenance.

Research Question 3: Is the training test suite's *size* related to patch overfitting?

To answer this question, we vary the training test suite size and observe the resulting overfitting rate. To achieve this, we randomly sample the black-box test suite (for the IntroClass dataset) and user's test suite (for the Codeflaws dataset) to obtain 25%, 50% and 75% of the suite as training tests, and use the resulting tests to guide repair. We vary the number of training tests, but keep the pass-fail ratio of tests in each version consistent. We repeat this experiment five times and aggregate the results for each repair technique. We also measure code coverage corresponding to the training test suites at various sizes as shown in Table 4.4. From the table, we can observe that the increase in test suite's size brings about higher code coverage.

Results. Tables 4.3(a) and 4.3(b) show results on the IntroClass and Codeflaws

Table 4.3: Overfitting rate of Angelix, CVC4, Enumerative, and SemFix when varying the number of tests used for training on IntroClass (top) and Codeflaws (bottom). ORate stands for overfitting rate.

(a) Overfitting by number of tests used, IntroClass.

	Angelix			CVC4			Enum			SemFix		
Subject	25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%
smallest	29/38	25/34	20/29	28/36	22/32	17/27	26/35	23/33	19/28	32/41	36/45	36/45
median	35/40	29/34	25/30	33/39	26/32	26/30	33/38	27/32	25/30	36/43	40/45	41/46
digits	8/8	6/6	7/7	8/8	6/6	6/6	8/8	6/6	6/6	9/9	12/12	12/12
ORate	84%	94%	78%	83%	77%	78%	83%	79%	78%	83%	86%	87.4%

(b) Overfitting by number of tests used, Codeflaws.

Angelix			CVC4			Enum			SemFix		
25%	50%	75%	25%	50%	75%	25%	50%	75%	25%	50%	75%
87/99	78/95	58/73	86/90	85/94	104/111	73/78	77/88	112/120	89/97	77/85	57/64
88%	82%	79.5%	96%	90.4%	94%	93.6%	87.5%	93%	92%	90.6%	86.4%

Table 4.4: Code coverage corresponding to training test suite's sizes of 25%, 50%, and 75% of original blackbox tests.

	Code Coverage		
Subject	25%	50%	75%
smallest	77.5%	80%	84%
median	73%	79%	82%
digits	73%	76%	77%
Codeflaws	79%	84%	85%

benchmarks, respectively. Interestingly, the overfitting rate fluctuation is very small. Table 4.3(a), shows that on IntroClass, using 25%, 50%, and 75% of black-box tests as training tests, Angelix has an overfitting rate of 84%, 94%, and 78%, respectively. This highlights an interesting trend: When training suite size increases, Angelix appears to generate fewer patches, but without a major change in overfitting rate. For example, considering *smallest* programs, Angelix generates 29, 25, and 20 non-overfitting patches when 25%, 50%, and 75% of black-box tests are used, respectively. We conclude that that it may be slightly more difficult to generate patches in response to higher-coverage test suites. However, as test suite coverage increases, overfitting rate does not appear to substantially decrease. Similar trends appear to apply to CVC4, Enumerative, and SemFix.

Table 4.3(b) shows the results on the Codeflaws benchmark. We can see that Angelix and SemFix follow the same trend as described above on the results on the IntroClass dataset. CVC4 and Enum, however, depict an opposite trend in terms of patch generation rate, wherein the number of generated patches increases with training test suite size.

These results are particularly interesting when contrasted with prior results characterizing overfitting for search-based repair [81]. [81] found that lower-coverage test suites posed a risk for search-based repair, leading to patches that were less likely to generalize. By contrast, our results for semantics-based repair do not show this relationship; test suite coverage overall may not influence the quality of semantics-based patches to the same degree they do in search-based techniques. As a result, semantics-based approaches may be safer to use than search-based techniques when only lower-coverage or lower-quality test suites are available.

Note that these semantics-based APR techniques generate repairs eagerly. That is, they generate one plausible repair at a time, and if that repair leads the program to pass all tests, it is returned without considering other candidates. Since there can exist many plausible patches that pass all tests, but are not necessarily correct (this has been empirically characterized for search-based techniques [59]), a potentially fruitful future direction for semantics-based APR may be to lazily generate a number of candidates using the synthesis strategy, and then employ an appropriate ranking function to heuristically rank candidates according to predicted correctness, combining various elements of both search-based and semantics-based approaches.

Research Question 4: How is the training test suite's *provenance* (automatically generated vs. human-written) related to patch overfitting?

Automatic test generation may provide a mechanism for augmenting inadequate test suites for the purposes of program repair. However, previous work assessing overfitting for search-based repair found that patch quality varied based on the origin (or *provenance*) of the training test suite on the IntroClass dataset. That is, the human and instructor-provided black box tests led APR techniques to produce higher-quality repairs, i.e., the ones that pass more held-out tests, than the automatically generated tests (generated by KLEE) [81]. We assess the same concern for semantic-based APR by comparing the quality of patches generated using the white-box (KLEE-generated) tests to those of the black-box (human-generated) tests from the IntroClass dataset. We only use IntroClass for this question since its held-out tests are automatically generated; the provenance of the held-out tests in Codeflaws is unspecified [82].

Results. The right-hand-side of Table 4.1(a) shows baseline patch results using the white-box tests for training; the right-hand side of Table 4.2(a) shows how many of those patches overfit. Angelix generates patches for 77 buggy programs using these test suites, including 37, 35, and 5 versions for subjects *smallest*, *median*, and *digits*, respectively. Of those, 31, 25, and 5 patches fail to generalize, respectively. Overall, when using white-box tests as training tests, Angelix generates patches with an overfitting rate of 72.7% on average. This is very slightly lower as compared to the rate for the black-box tests, seen in Research Question 3 (75% versus 72.7%).

This result on semantics-based repair is particularly interesting as compared to that of the search-based repair case [81]. Smith et al. [81] found that overfitting in search-based repair is worse when using whitebox tests for training. These results suggest that although automated test generation may not help with search-based repair, it could be particularly useful in helping semantics-based repair, i.e., Angelix to mitigate the risk of overfitting. As above, lower-quality test suites may pose a smaller risk to the output quality of this technique type.

By contrast, the performance of CVC4 and Enumerative suffers when using the white-box as compared to the black-box tests. CVC4 and Enumerative can only generate non-overfitting patches for 4 and 3 versions of smallest, respectively. This indicates a very high overfitting rate (around 95%). The performance of SemFix is almost the same when either using black-box or white-box tests as training tests (overfitting rate of around 87%). Recall that our experiments use one set of test for training, e.g., whitebox tests, and the other for testing, e.g., blackbox tests. Thus, one possible reason for worse performance of CVC4 and Enumerative when training using whitebox tests as compared to blackbox tests could be that the blackbox tests are possibly more comprehensive than whitebox tests. This implies that when testing using backbox tests, plausible repairs – which pass a certain set of tests but do not generalize to other test set – could be more easily detected as compared to testing using whitebox tests.

The differences between the results on Angelix versus that of other synthesis engines, i.e., CVC4, Enumerative, and SemFix could be due to both the nature of the underlying synthesis techniques and the datasets used for experiments. Angelix's synthesis engine attempts to generate minimal repair, which renders the fixed program to be minimally syntactically different from the original buggy one, by using MaxSMT. Other techniques including CVC4, Enumerative, and SemFix do not constrain the relationship between the repaired program and the original buggy one, and thus may generate patches involving larger changes that make the repaired program very different from the original buggy one. In fact, datasets used for our experiments require small changes to fix bugs. Thus, Angelix-generated (minimal) patches may be more likely to be correct, while patches requiring larger changes generated by other techniques may be more likely plausible – which can pass certain set of tests but do not generalize. Overall, there remains a need to improve automated test generations before it can be used across the board for automatic repair, and to understand the source of this quality discrepancy.

Table 4.5: Overfitting rate when using all training tests for specification inference for IntroClass (top; we omit syllables and checksum, for which no patches were generated) and Codeflaws (bottom).

(a) Overfitting on IntroClass

Subject	Angelix	CVC4	Enum	SemFix
smallest	10 / 20	27 / 39	27 / 39	12 / 19
median	18 / 22	13/ 20	24 / 32	18 / 23
digits	5 / 5	3 / 3	3 / 3	5 / 5
Overfitting	70.2%	69.4%	73%	74.5%

(b) Overfitting on Codeflaws

	Angelix	CVC4	Enum	SemFix
	102 / 126	102 / 108	79 / 84	100 / 112
Overfitting	81%	95%	94%	89.3%

4.3.4 Tunable technique parameters

Our next two research questions concern the relationship between patch generation success and quality and (7) number of tests used for specification inference and (8) the Angelix group size feature.

Research Question 5: What is the relationship between the number of tests used for specification inference and patch generation success and patch quality?

Theoretically, the more tests used for specification inference, the more comprehensive the inferred specifications, which may help synthesis avoid spurious solutions.[8] Thus, we investigate the relationship between the number of considered tests and patch generation and quality for all considered techniques.

IntroClass results. We use black-box tests as training tests, and white-box tests as held-out tests to answer this question, and instruct the inference engine to use *all* available tests for specification inference. The top of Table 4.5 shows results. Angelix generates 47 patches, of which 33 do not fully generalize, indicating an overfitting rate of 70.2% on average. As compared to the results from Research Question 3, in which we use Angelix's default setting

[8]Recall the explanation in Section 4.2.3 on the number of tests used for specification inference.

(starting with two tests), the overfitting rate is slightly reduced (from 75% to 70.2%). CVC4 and Enumerative generate patches with overfitting rate of 69.4% (43 incorrect patches over 62 generated patches), and 73% (54 incorrect patches over 74 generated patches), on average, respectively. The effect on overfitting rate is more dramatic for these approaches. CVC4's overfitting rate decreases, from 80.3% to 69.4%. Similarly, SemFix's overfitting rate decreases from 90% to 74%: it generates 35 incorrect patches over 47 generated patches. Overall, these results suggest that using more tests for specification inference helps semantics-based program repair to mitigate overfitting, supporting our hypothesis.

Codeflaws results. We use tests that are available to users as training tests, and tests that are available only to the contest's committee as held-out tests. The last row of Table 4.5 shows results. Angelix generates 126 patches, of which 102 do not generalize, indicating an overfitting rate of 81%. Compared to the results shown in Table 4.2(b) in research question 3, which uses two tests for specification inference, Angelix generates more patches (increased from 81 to 126 patches) but escalates the overfitting rate (from 54% to 81%). The similar trend can be seen for CVC4, Enumerative, and SemFix. This results actually contradict our hypothesis.

We believe that this fact could be due to a combination of several reasons. When using all repair (training) tests for inference task, once a solution is synthesized consistent with the specifications, it satisfies the whole repair test suite and thus regarded as a patch. Therefore, if the repair test suite is weak enough to allow such a situation, it results in an increase in patch generation rate. However, the in-comprehensiveness of the repair test suite also brings about a reasonably high probability of the overfitting rate since the generated patches may not generalize. In fact, the size of repair test suite in the Codeflaws benchmark is quite small (only 3 tests on average), while the held-out test suite's size is much larger (40 tests on average) [82].

Table 4.6: The overfitting rate of Angelix, CVC4, Enumerative, and SemFix in subject programs from IntroClass (omitting syllables and checksum, for which no patches were generated) and Codeflaws, when group size is set to three and four, respectively.

(a) Overfitting on IntroClass

	Size 3			Size 4		
Subject	Angelix	CVC4	Enum	Angelix	CVC4	Enum
smallest	8 / 20	10 / 19	10 / 20	12 / 18	10 / 19	11 / 21
median	18 / 24	17/ 23	18 / 23	17 / 23	17 / 23	19 / 25
digits	5 / 5	5 / 5	5 / 5	5 / 5	5 / 5	5 / 5
Overfitting	63.3%	68%	69%	74%	68%	68.6%

(b) Overfitting on Codeflaws

	Size 3			Size 4		
	Angelix	CVC4	Enum	Angelix	CVC4	Enum
	75 / 87	84 / 86	57 / 60	89 / 108	43 / 44	48 / 49
Overfitting	86.2%	98%	95%	82.4%	98%	98%

Research Question 6: How does the number of fault locations grouped together affect patch generation rate and overfitting?

The second tunable feature we study is the effect of grouping faulty locations. The larger the group, the expressions considered for repair at once. We observe the behaviors of different repair techniques when the group size is set to 3 and 4, respectively. We note that SemFix is left out in this research question since it is not able to fix multi-line bugs [68].

Results. Table 4.6 shows the results on both IntroClass and Codeflaws benchmarks. Overall, the number of generated patches and the overfitting rate when group size varies only slightly between the group sizes. On IntroClass, Angelix generates 49 and 46 patches, with overfitting rates of 67% and 74%, when group size is set to 3 and 4, respectively. As compared to research question 3, which uses default group size of two, the number of generated patches substantially decreases, e.g., from 81 to 49 and 46 for Angelix. CVC4 and Enumerative show a similar trend. We hypothesize that increasing the number of likely-buggy locations being fixed proportionally enlarges the search space, and subsequently makes it harder to generate patches.

The same trend generally holds on the Codeflaws dataset, with an interesting exception for Angelix. Angelix generates more patches (87 vs 108), while reducing the overfitting rate slightly (86% vs 82%) when group size is varied from 3 to 4. This shows that Angelix's ability in fixing multi-line bugs is potentially helpful in this case.

4.3.5 Examples from the IntroClass dataset

We now present and discuss several examples that may provide deeper insights into the overfitting issue for semantics-based APR.

```
 1  void median(int n1, n2, n3) {
 2    int small;
 3    if (n1 < n2){
 4      small = n1;
 5      if (small > n3)
 6        printf("%d\n", ANGELIX_OUTPUT(int, n1, "stdout"));
 7      else if (n3 > n2)
 8        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
 9      else
10        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
11    } else {
12      small = n2;
13      if (small > n3)
14        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
15      else if ( n3 > n1 )
16        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
17      else
18        printf("%d\n", ANGELIX_OUTPUT(int, n1, "stdout"));
19    }
20  }
```

Figure 4.3: Example of a buggy *median* program (simplified slightly for presentation). The buggy line is shaded in blue at line 15. The *ANGELIX_ OUTPUT* macro is explicitly required Angelix instrumentation; it indicates output variables to Angelix.

Figure 4.3 shows an example of a buggy *median* program. The goal of a *median* program is to identify the median value between three integer inputs. The buggy line in our example is colored blue, at line 15. We now consider Angelix-generated patches for this example program using 25% and 50% of

```
11     else {
12      small = n2;
13      if ( small > n3 )
14        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
15      else if ( n1 > n1 )
16        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
```

(a) Patch created using 25% of black-box tests, changing lines 17 and 19 of the original program, shaded in red.

```
11    else {
12      small = n2;
13      if ( small > n3 )
14        printf("%d\n", ANGELIX_OUTPUT(int, n2, "stdout"));
15      else if ( n1 > n3 )
16        printf("%d\n", ANGELIX_OUTPUT(int, n3, "stdout"));
```

(b) Patch created using 50% of the black-box tests, changing lines 17 and 19 of the original program, shaded in red.

Figure 4.4: Patches generated by Angelix for the program in Figure 4.3 using 25% of the black-box tests (top) and 50% of the black-box tests (bottom) as training tests. Line numbers are aligned with those in Figure 4.3.

the black-box tests for training. Figure 4.4(a) shows the Angelix patch for the program in Figure 4.3 using 25% of black-box tests for training. The patch considers the expressions at lines 13 and 15, respectively, for repair (colored red). Line 13 remains unchanged, while the true buggy condition at line 15 is changed. This patch overfits, such that the resulting program does not pass all held-out tests (such as the test $\{n1 = 8,\ n2 = 2,\ n3 = 6\}$).

Test ID	n1	n2	n3	Expected State
#3	6	2	8	(L13 → false) ∧ (L15 → false)
#5	8	2	6	(L13 → false) ∧ (L15 → true)

Figure 4.5: Specifications inferred by Angelix for the example in Figure 4.3 using 50% of black-box tests for training. (The first row shows the specification inferred using only 25% of the tests for training). The first column shows the test id. The next three columns show values of *n1*, *n2*, and *n3*, respectively. The last column shows the expected states at different lines given the input values. For example, (L13 → false) ∧ (L15 → false) in test id #3, means the expected state of the *if-conditions* at lines 13 and 15 are both false.

To better understand this issue, consider the specification inferred by Angelix that lead to this erroneous patch. The first line of Figure 4.5 shows the specification that lead to this patch, produced on a test with input values of *n1*

```
1      if (num1 > num2) {...}
2      else {
3        big = num2;
4   -    small = num2;
5   +    small = 6; // by Angelix
6   +    small = num1 // by SyGuS
```

Figure 4.6: Patches generated by Angelix's synthesis engine, and SyGuS engines for a *median* program.

= 6, *n2* = 2, and *n3* = 8. This specification indicates that this test would pass if the states of the *if-conditions* at lines 13 and 15 are both *false*, which the patch in Figure 4.4(a) satisfies. This shows the danger of weak specifications.

Returning to the example program (Figure 4.3), consider adding an additional test, with associated inferred specification, shown in the second line of Figure 4.5. Adding this test to the training set leads Angelix to find the patch shown in Figure 4.4(b), which is generally correct in the way it changes the logic of the *if-condition* at line 15. In this case, increasing the number of training tests (from 25% to 50% of black-box tests) provided a huge benefit: This patch fully generalizes to the held-out tests, and it better matches our intuition. One conclusion is that additional tests can help guide synthesis to a better repair, which is especially satisfying in this case, where only two total are required.

Figure 4.6 shows patches generated by Angelix's synthesis engine and SyGuS engines for another *median* program. Angelix's patch replaces line 4 with line 5; the SyGuS engines (including CVC4 and Enumerative) replace line 4 with line 6. Angelix's generated patch is incorrect; the SyGuS- generated patch is correct. Angelix's synthesis engine does not force generalization, where a generalized solution involves as few constants as possible [27]. SyGuS engines, on the other hand, are more flexible in forcing generalization by simply emphasizing permitted constants after variables in its grammar. This suggests a straightforward strategy to improve the generality of patches produced by such techniques.

Figure 4.7 shows an example of an overfitting patch for the *smallest* subject

```
 1       ...
 2  -   if((a < b) && (a < c) && (a < d))
 3  +   if(((a < d) && (a < d)))
 4          printf("%d\n",ANGELIX_OUTPUT(int,(int) a, "stdout"));
 5  -   else if ((b < a) && (b < c) && (b < d))
 6  +   else if ((((b < a) && (b < c)) && (b < d)))
 7          printf("%d\n",ANGELIX_OUTPUT(int,(int) b, "stdout"));
 8      else if ((c < a) && (c < b) && (c < d))
 9          printf("%d\n",ANGELIX_OUTPUT(int,(int) c, "stdout"));
10  }
```

Figure 4.7: Example of an overfitting patch for the *smallest* subject program, generated by Angelix when using all black-box tests as training tests.

program, generated by Angelix when using all black-box tests as training tests. The goal of the *smallest* program is to return the smallest of three integer numbers. The patch generated by Angelix replaces line 2 with line 3, loosening the *if-condition*. As with our prior example, this patch clearly overfits to a particular set of tests. This example demonstrates that overfitting can occur even when a full set of black-box tests is used.

This bug would likely benefit from multi-location patch that adds equality signs to each of the conditions in lines 2, 5 and 8. Yet, Angelix's ability to generate multi-location patch does not help in this case. Angelix's ad-hoc approach to deciding how many buggy locations to consider, and how to group them, could be improved with stronger heuristics, more accurate fault localization, or more precise dataflow information to better group the three implicated conditions. Generally, however, these results call for the development of stronger or alternative synthesis engines that are more resilient to overfitting.

4.3.6 Qualitative study on real-world bugs

Our results in preceding sections describe program repair as applied to small programs with two independent test suites. We now present qualitative results assessing the performance of different synthesis engines on defects in large, real-world programs. Automatically generating independent full-coverage test suites on real-world programs is prohibitive. As such, we employ a stricter proxy to assess the correctness of machine-generated patches: A machine-generated patch is considered correct if it is equivalent to the patch submitted by developers. Two patches are considered equivalent if: (1) they are syntactically identical, or (2) one patch can be transformed into the other via basic syntactic transformation rules. For example, `a || b` and `b || a` are considered equivalent if no observable side effects occur when evaluating either a or b, e.g., exceptions thrown, modifications to global variables, etc. We choose syntactic transformations as the proxy for correctness validation because checking for semantic equivalence is a hard problem, and undecidable in general. A semantic equivalence check may involve deep human reasoning, which may be subjective. We thus use syntactic equivalence to ease the validation process and avoid subjectivity, although it may be overly strict in certain cases.

To ease this manual process, we require a transparent baseline, in that the developer-submitted patches (ground truth) must be sufficiently concise and transparent to support a manual patch-equivalence check. Unfortunately, existing benchmarks (such as ManyBugs [53]) often include changes of multiple lines, complicating manual correctness assessment. To this end, we reuse a benchmark consisting of nine real-world bugs in large programs (such as the Common Maths library, consisting of 175 kLOC), and tool named JFIX, from our work in [45]. JFIX adapts the Angelix specification inference engine to Java programs [45], and uses the synthesis engine of Angelix, CVC4, and Enumerative to synthesize repairs conforming to the inferred specifications. We omit SemFix, because it generally does not scale to these programs [68].

Table 4.7 shows results. Note that this study extends our previous work [45] by further studying the effect of a majority voting scheme. That is, we check whether we can choose a correct patch from a set of patches suggested by the synthesis engine by employing majority voting (i.e., patches that are generated by most of the synthesis engines are chosen). Machine-generated patches that are equivalent to patches submitted by developers are indicated by "✔", and

"✗" indicates otherwise in Table 4.7.

As Table 4.7 shows, Angelix, Enumerative and CVC4 can complement one another. There are no bugs for which all three techniques produce equivalent patches, but for all bugs, at least one technique does succeed. For example, there are three bugs for which Enumerative and CVC4 can generate patches equivalent to their developer counterparts, where Angelix does not. This shows that the nature of each repair technique may lead to different kind of patches, suggesting that employing an agreement (majority voting) on generated solutions between different synthesis engines may increase confidence when choosing a correct patches. The high level idea is that the more different synthesis engines agree on a solution, the higher the confidence that the solution is correct. For example, CVC4 and Enumerative generate the same (correct) solutions for *SFM* and *EWS*, while Angelix generates a different solution. We note that this majority voting method can still lead to incorrect patches. For example, Angelix generates a correct patch for *Qabel*, but CVC4 and Enumerative both generate the same incorrect patch. Thus, if majority voting is employed, it leads to an incorrect patch in this case. However, as the overall results indicate, the majority voting method is more effective than any individual technique in identifying correct patches.

Table 4.7: Real bugs collected from real-world software. Column "Rev" shows the revisions that fix the bugs. Column "Type" shows the bug types: "I" denotes method call, "II" denotes arithmetic. Column "Time" indicates the time required (in seconds) to generate the repair ("NA" denotes not available). Column "Dev" indicates whether a generated repair is equivalent to the repair submitted by developers. The "✔" denotes equivalent, and the "✗" denotes otherwise. The "Majority Voting" column shows the result of using majority voting method to choose correct patches.

Project	Rev	Type	Angelix		Enum		CVC4		Majority Voting
			Time	Dev	Time	Dev	Time	Dev	
Math	09fe	II	23s	✔	26s	✔	36s	✔	✔
	ed5a	I & II	168s	✔	NA	✗	NA	✗	✔
Jflex	2e82	II	NA	✗	70s	✔	72s	✔	✔
Fyodor	2e82	II	20s	✔	19s	✔	31s	✔	✔
SFM	5494	II	12s	✗	10s	✔	13s	✔	✔
EWS	299a	I	NA	✗	14s	✔	258s	✔	✔
Orientdb	b33c	II	20s	✔	22s	✔	NA	✗	✔
Qabel	299c	II	37s	✔	22s	✗	23s	✗	✗
Kraken	8b0f	II	12s	✔	13s	✗	15s	✗	✗

4.4 Conclusions

In this work, we perform the first study on overfitting issue in semantics-based APR. We show that semantics-based APR techniques do indeed produce patches that overfit. We further study the nature behind the overfitting in semantics-based APR by assessing the relationships between test suite coverage and provenance, number of failing tests, and semantics-specific tool settings and overfitting. Particularly, we find that in some cases results are consistent with those found for search-based approaches, while in other cases results are interestingly inconsistent, e.g., using whitebox tests as training tests reduces the overfitting rate of semantics-based repair, while increases the overfitting rate of search-based repair. We also present several case studies of overfitting patches produced by semantics-based APR techniques, with implications and observations for how to improve them. For example, we observe that one possible source for overfitting in semantics-based APR could be due to the "conservativeness" of the underlying synthesis engine, that returns the first solution found (without consideration of alternatives). That is, each synthesis engine used in our experiments returns only one solution (repair) once it is found. However, the solution returned may just be a plausible one, which is not the correct repair. To obtain the correct repair, one may need to go through all solutions that possibly occur in the search space and rank them by the likelihood of being correct to find the best one. Also, to mitigate overfitting in semantics-based APR, we substantiate that using multiple synthesis engines could be one possible approach, as mentioned in [44]. We also plan to develop a specification inference technique, e.g., specification mining techniques such as SpecForce [43], that can infer a stronger specifications to help better capture the semantics of the program under repair. Another future direction is to use machine learning techniques to automatically classify defect types, e.g., [85], which could help deal with each bug type more effectively.

Chapter 5

Syntax- and Semantic-Guided Repair Synthesis

Previous chapter highlighted that semantics-based APR is no exception at all to the overfitting issue, often due to inferior synthesis techniques that do not cope well with the weak specifications extracted via test cases. This motivates the need of stronger synthesis techniques that can overcome the overfitting problem. This chapter presents *S3* – a scalable semantics-based APR technique that is capable of synthesizing generalizable repairs. The main novelty of *S3* is three-fold: (1) a systematic way to constrain the syntactic search space via a domain specific language, (2) an efficient enumeration search strategy, (3) a number ranking features to effectively rank solutions based on the likelihood of being correct.

5.1 Introduction

Bug fixing is notoriously difficult, time-consuming, and costly [84, 12]. Hence, automating bug repair, to reduce the onerous burden of this task, would be of tremendous value. Automatic program repair has been gaining ground, with substantial recent work devoted to the problem [65, 66, 95, 58, 60, 96, 52, 37, 48, 47, 45, 14], inspiring hope of future practical adoption. One notable line of work in this domain is known as *semantics-based* program repair, most recently embodied in Angelix [66]. This class of techniques uses semantic analysis (typically dynamic symbolic execution) and a set of test cases to infer behavioral specifications of the buggy code, and then program synthesis to construct repairs that conform to those specifications. Such approaches have recently been shown to scale to bugs in large, real-world software [66].

Although scalability has been well-addressed, one pressing concern in program repair is patch quality, sometimes quantified in terms of patch *overfitting* or *generalizability* [81]. Generated repairs can sometimes overfit to the tests used for repair, and fail to generalize to a different set of tests. This may be caused by weak or incomplete tests, or even simply the nature of the repair technique [81, 36]. Various repair approaches have been shown to suffer from overfitting, including GenProg [52], RSRepair [73] and SPR [58]. Semantics-based approaches like Angelix [66], are no exception to this issue, as partially shown in recent studies [49]. Overfitting, and patch quality generally, remains a challenging problem in the program repair field.

One reason for patch overfitting is that the repair search space is often sparse, containing many plausible solutions that can lead the buggy program to pass a given test suite, but that may still be judged incorrect [59]. One way to tackle overfitting is thus to constrain the search space to patches that are more likely to generalize. Other strategies for increasing the quality of output patches include higher-granularity mutation operators [36], anti-patterns [83], history-based patterns [50], feedback from execution traces [21], or document analysis [95]. Angelix [66] eagerly preserves the original syntactic structure of the buggy program via PartialMaxSMT-based constraint solving [65] and component-based synthesis [32]. However, such enforcement alone may not be enough [21]. Furthermore, incorporating other strategies or criteria into a constraint-based synthesis approach is non-obvious, since doing so typically requires novel, and often complicated constraint encodings (this problem has been pointed out by others, see, e.g., Chapter 7 of [27] or Section 2 of [83]). This motivates the design of a new repair synthesis technique that can consolidate various restrictions or patch generation criteria, enabling an efficient search over a constrained space for potentially higher-quality patches.

We present S3 (Syntax- and Semantic-Guided Repair Synthesis), a new, scalable repair synthesis system. S3 addresses the challenge of synthesizing generalizable patches via our novel design of three main components: (1) An underlying domain-specific language (DSL) that can systematically customize and constrain the syntactic search space for repairs, (2) An efficient enumeration-based search strategy over the restricted search space defined by the DSL to find solutions that satisfy correctness specifications, e.g., as induced by test suites, and (3) Ranking functions that serve as additional criteria aside from the provided specifications to rank candidate solutions, to prefer those that are more likely to generalize. Our ranking functions are guided by the intuition that a correct patch is often syntactically and semantically proximate to the original program, and thus measure such syntactic and semantic distance between a candidate solution and the original buggy program. Unlike other constraint-based repair synthesis techniques, our framework is highly customizable by design, enabling the easy inclusion of new ranking features — its design is inspired by the programming-by-examples (PBE) synthesis methodology [27].

Given a buggy program to repair and a set of test cases (passing and failing), S3 works in two main phases. The first phase automatically localizes a repair to one or more target repair expressions (e.g., branch condition, assignment right-hand-side, etc.). S3 runs dynamic symbolic execution on the test cases to collect *failure-free* execution paths through the implicated expressions. It then solves the collected path constraints to generate concrete expression values that will allow the tests to pass. These specifications, expressed as input- and desired-output examples, are input to the synthesis phase. The synthesis phase first constrains the syntactic search space of solutions via a DSL that we extend from SYNTH-LIB [4]. Our extension allows it to specify a starting *sketch*, or an expression that gives S3 clues about what possible solutions might look like. Here, the sketch is the original buggy expression under repair. Next, S3 forms a solution search space of expressions of the same size as the sketch. Finally, it ranks candidate solutions via a number of features that approximate the syntactic and semantic distance to the specified sketch. If S3 cannot find any solution of the same size as the sketch, it investigates expressions that are incrementally smaller or larger than the sketch, and repeats the process.

We evaluate S3 by comparing its expressive power and the quality of the patches it generates to state-of-the-art baseline techniques (Angelix [66]; and Enumerative [4], and CVC4 [77] two alternative syntax-guided synthesis approaches), on two datasets. The first dataset includes 52 bugs in small programs, a subset of the IntroClass benchmark [53] translated to Java [22].[1] The IntroClass dataset contains only small programs, but provides two high-coverage test suites for each, allowing an independent assessment of repair quality. The second dataset includes 100 large real-world Java bugs that we collected from GitHub. We focus on Java, and build a new dataset of real-world Java bugs, for several reasons. First, Java is the most popular and widely-used programming language, and its influence is growing rapidly.[2] Second, a realistic, real-world dataset with transparent ground truth — fixes submitted by developers — can simplify the critical process of assessing the correctness of fixes generated by program repair tools in the absence of two independent, high-quality test suites. Existing benchmarks often include bug fixes with many changed lines, which can include tangled changes such as new features or code refactoring [30]; even curated datasets such as Defects4J [34] contain many changes involving a large number of lines. This complicates evaluation of

[1]We use the subset of IntroClass to which our repair tools can apply, given their applicability to strictly integer and boolean domains.

[2]http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

generated patch correctness. Our dataset is restricted to bugs whose fixes involve fewer than five lines of code, alleviating the risk of tangled code changes. As many current state-of-the-art repair tools target bugs that require only a small number of changed lines [65, 66, 60], our dataset is sufficient for assessing current research.

We assess the quality and correctness of generated repairs in several ways. For the IntroClass bugs, we assess correctness on independent, held-out test suites (those provided with the benchmark, as well as additional tests we generate), separate from those used to guide the repair. We use the developer-provided patches as ground truth for the 100 real-world bugs. For these bugs, we consider a generated patch correct if it is either (1) *syntactically* identical to the developer-provided patch, or (2) semantically equivalent via some (basic) transformations. On both datasets, S3 substantially outperforms the baselines. S3 generates correct patches for 22 of 52 bugs from the first dataset; Angelix, Enumerative, and CVC4 can generate correct patches for 7, 1, and 1 bug(s), respectively. On the large real-world dataset, S3 generates correct patches for 20 out of 100 bugs, while Angelix, Enumerative, and CVC4 can only generate correct patches for 6, 6, and 5 bugs, respectively.

In summary, our novel contributions include:

- We present S3, a scalable repair synthesis engine that is geared towards synthesizing generalizable repairs.
- We propose a novel combination of syntax- and semantic-guided ranking features to effectively synthesize high-quality repairs. New features along these lines can be straightforwardly integrated into S3, by design.
- We present a large scale empirical study on the effectiveness of different synthesis techniques in semantics-based program repair context. S3 substantially outperforms the baselines in terms of generated repair quality.
- We present a dataset consisting of several bugs from large real-world software

with transparent ground truth, which can enable confident evaluation of machine-generated patch correctness.

- We release source code for S3 and the aforementioned dataset, along with all results, in support of open science.[3]

The rest of the chapter is structured as follows. Section 5.2 describes a motivating example, followed by Section 5.3 explaining our approach. Section 5.4 describes our experiments, results, and observations. Section 5.5 concludes.

5.2 Motivating Example

We begin by motivating our approach and illustrating its underlying insight by way of example. Figure 5.1 shows changes made to address a bug in the Closure compiler at revision *1e070472*. The bug lies in the if-condition expression at lines 3–4; the developer-submitted fix is depicted at lines 5–6. This bug can be repaired by simply changing `charno < sourceExcerpt.length()` to `charno <= sourceExcerpt.length()`, while the rest of the condition expression remains unchanged.

```
1 if (sourceExcerpt != null) {
2   ...
3   -if (excerpt.equals(LINE) && 0 <= charno
4   -    && charno < sourceExcerpt.length()) {
5   +if (excerpt.equals(LINE) && 0 <= charno
6   +    && charno <= sourceExcerpt.length()) {
7   ...
8 }
```

Figure 5.1: A bug in Closure compiler, revision *1e070472*. The bug is at lines 3–4. The developer fix is shown on lines 5–6; it turns a `<` to a `<=` in the second line of the if condition.

Figure 5.2 shows example input and desired-output examples extracted for this bug at the buggy if-condition on two failing test cases. For each test run, the input includes runtime values of variables and method calls at the

[3]https://xuanbachle.github.io/semanticsrepair/

buggy lines, while the output is the value of the branch condition for the buggy lines that would cause the test to pass. For example, for test 1, the input includes runtime values for method calls `excerpt.equals(LINE)` and the variable `charno`. The desired output of the branch condition is `true`. These input-output examples constitute incomplete specifications for each buggy line considered in the program; although they are incomplete, they are scalably and automatically derivable from provided test cases.

	Input			**Desired**
		(M1)	*(M2)*	
Test	`charno`	`excerpt.equals(LINE)`	`sourceExcerpt.length()`	**Output**
A	7	`true`	7	`true`
B	10	`true`	10	`true`

Figure 5.2: Input-output examples for both variables and conditions, extracted for the Closure compiler bug described in Figure 5.1. We use M1 and M2 to refer to the conditions in columns 3–4 in subsequent exposition. The last column represents the desired output of the overall branch decision.

Given these specifications (examples), the space of possible satisfying solutions is large, and contains many undesirable options, such as `excerpt.equals(LINE)`, `excerpt.equals(LINE)|| 0 < charno`, both of which, among others, would lead to the desired outputs on the considered expressions. Such solutions, if returned by a repair synthesis engine, create low-quality, *overfitting* repairs that lead the program to pass all provided tests but are not correct. In fact, Angelix [66] generates an overfitting repair for this bug, substituting `0 < charno` for the entire if-condition expression on lines 3–4 (Section 5.4 provides details on our straightforward port of Angelix to Java). This repair is quite different from the original expression both syntactically (despite Angelix's use of constraints to enforce minimal syntactic differences from an original expression) and semantically. The generated condition is indifferent to values of `excerpt.equals(LINE)` and `sourceExcerpt.length()`, substantially weakening the branch condition with respect to the original buggy version.

These observations inform insights that can be used to filter trivial solutions. In this case, the correct solution is syntactically and semantically close to the original buggy expression. Fusing syntactic and semantic measures of proximity can help rank the solution space to favor those that are more likely to be correct. Our approach, S3, estimates these distances in several ways to constrain the syntactic solution synthesis space, increasing the likelihood or producing a generalizable patch (see Section 5.3.2.3). For the example in Figure 5.1, S3 synthesizes a patch that is identical to the one submitted by the developer.

5.3 Methodology

S3 works in two main phases. Given a buggy program and a set of test cases, the first phase (Section 5.3.1) localizes potentially buggy program locations and, for each buggy location, extracts input and desired output examples that describe passing behavior. The extracted examples are input to the second phase (Section 5.3.2), which synthesizes repairs that satisfy and also generalize beyond the provided examples.

5.3.1 Automatic Example Extraction

S3 first uses fault localization to identify likely-buggy expressions or statements in the buggy program. *S3* runs the test cases and uses Ochiai [3] to calculate suspiciousness scores that indicate how likely a given expression or a statement is to be buggy. *S3* iterates through each identified buggy location (or group of locations in the case of multi-location repair), to extract input-output examples via a selective, dynamic symbolic execution [15].[4] For each buggy location, *S3* inserts a symbolic variable to represent/replace the expression at the selected location. It then invokes test cases on the instrumented programs to collect

[4]For simplicity, we describe the process with respect to a single location; it extends naturally, by installing symbolic variables at multiple locations at once.

path conditions that do not lead to runtime errors such as assertion errors, array index out of bound errors, etc. Solving these failure-free execution paths returns concrete values of symbolic variables that then can serve as input-output examples. We implement selective symbolic execution procedure on top of Symbolic PathFinder (SPF) [72].

For example, consider the buggy code snippet in Figure 5.1. *S3* identifies that the if-condition at lines 3-4 may be buggy. *S3* then replaces the buggy if-condition with a symbolic variable α, making the if-condition becomes "`if(` α`)`". *S3* runs dynamic symbolic execution on the instrumented program using the provided test cases to collect failure-free execution paths, runtime variable values, and method calls involved in the buggy location. Solving the collected path conditions returns the values in the output column of Figure 5.2, corresponding to desired values of the symbolic variable α.

Although this phase shares the same spirit as the specification inference step in Angelix [66], there are key differences. Angelix infers specifications by solving models of the form $pc \wedge O_a = O_e$, where pc is a path condition produced by symbolic execution of a test, O_a is the actual output, and O_e is the expected output that is typically manually provided by a user.[5] The models capture the idea that if the expected output matches the actual concrete test output, the corresponding path condition is a test-passing path. Solving all test-passing paths returns specifications that lead all tests to pass. This process, however, can be tedious and error-prone, since it usually requires users to instrument output variables manually. For instance, if the output is a large array of many elements, users must give all expected outputs for all the elements of the array.

S3 extracts examples in an automated manner by building on SPF [72] automatic JUnit-test interpretation abilities. For a location i, *S3* extracts examples by solving models of the form $pc \wedge$ *no errors*. pc is the path condition: $\bigwedge_{j=1}^{i} pc_j$. The "*no errors*" notation means that the conditions describe paths

[5]We refer readers to the Angelix manual: `https://github.com/mechtaev/angelix/blob/master/doc/Tutorial.md`

that are guaranteed to not yield assertion errors (as described above). If the path condition *pc* yields an assertion error, *S3* automatically discards that path. In another case, if an array-out-of-bound error happens, *S3* pops the latest pc_i leading to the error, keeping previous ones: $\bigwedge_{j=1}^{i-1} pc_j$. This frees *S3* users from manual effort, while guaranteeing that the examples are still failure-free.

5.3.2 Repair Synthesis from Examples

Examples extracted in the previous phase are input as correctness specifications to the repair synthesizer. The goal of the synthesizer is to inductively construct a solution that satisfies and also generalizes beyond the provided specifications. This synthesis procedure is composed of three main parts: (1) a domain-specific language (DSL), (2) a search procedure, and (3) ranking features. We begin with an overview, and detail each component subsequently.

We start with a DSL (extended from SYNTH-LIB [4]) over the integer and boolean domains. Given a background theory T permitted by the DSL, let u be the original buggy expression, ϕ a formula over the vocabulary of T representing the correctness specifications (input-output examples), and L a set of expressions over the vocabulary of T of the same type as u. A *candidate fix* is an expression $e \in L$ such that $\phi[u/e]$ is valid modulo T.

Our algorithm then systematically enumerates all candidate fix expressions, considering them in ranked order. The ranking is performed by a set of N ranking functions $r_i (1 \leq i \leq N)$, each of which measures the distance between two expressions e_1 and e_2 of the same type. These ranking features estimate the syntactic and semantic distance between a candidate fix and the original buggy expression. The intuition is that expressions that are closer to the buggy program are more likely to constitute high-quality repairs.

Note, however, that the size of L (the search space) is often too large to be truly exhaustively enumerated. For practical purposes, we *greedily* favor can-

$$\begin{array}{ll}
\mathit{IntExpr} & ::= \mathcal{N} \mid \mathit{Var} \mid \mathit{IntExpr}\ \mathit{BinOp}\ \mathit{IntExpr} \\
\mathit{BoolExpr} & ::= \mathit{IntExpr}\ \mathit{RelOp}\ \mathit{IntExpr} \mid \mathit{BoolExpr}\ \mathit{LogOp}\ \mathit{BoolExpr} \\
 & \quad \mid \mathtt{true} \mid \mathtt{false} \mid \mathit{Var} \mid \neg \mathit{BoolExpr} \\
\mathit{RelOp} & ::= \ > \ \mid \ < \ \mid \ \leq \ \mid \ \geq \ \mid \ = \\
\mathit{LogOp} & ::= \wedge \mid \vee \mid \ = \qquad \mathit{BinOp} \ ::= \ + \ \mid \ -
\end{array}$$

Figure 5.3: Simplified SYNTH-LIB grammar used in *S3*.

didate expressions of similar size and syntax to the original buggy expression. As described in Section 5.3.2.2, we systematically partition the search space, enabling different heuristics to be built without difficulty.

Algorithm 2 presents pseudocode for *S3*. At a high level, the search procedure enumerates all expressions in the grammar at a certain expression-size range (Line 4). *S3* finds all candidate enumerated expressions that are consistent with the specifications (Line 7). Each candidate is assigned a ranking score by calculating the distance between it and the original buggy expression (Lines 8); candidates are sorted by score (Line 12). The process returns the solution in L with the smallest distance, if $L \neq \emptyset$ (Line 14). Otherwise, it continues until all expression size ranges have been exhausted (Line 3). *S3* starts enumerating at the size of the original buggy expression (Line 2), and modifies the size range accordingly up to a bound b (Line 3). The original buggy expression and its size are made available to the synthesis procedure through our "*sketch*" extension to the SYNTH-LIB syntax (Section 5.3.2.1).

We next explain the DSL in detail (Section 5.3.2.1), the enumeration-based search procedure (Section 5.3.2.2), and the ranking features that we propose for the program repair domain (Section 5.3.2.3).

5.3.2.1 Domain-Specific Language via SYNTH-LIB

We extend SYNTH-LIB [4] to systematically constrain *S3*'s search space. We choose SYNTH-LIB for three reasons:

(1) Balanced Expressivity. SYNTH-LIB is adequately expressive for various tasks in the program repair domain, while still sufficiently restrictive to

Algorithm 2: Enumeration-based synthesis procedure

```
Input :
        u                              ▷ Original buggy expression
        $\phi$                              ▷ Correctness specifications
        G                              ▷ SYNTH-LIB grammar (extended)
        R                              ▷ Set of ranking features
        b                              ▷ Synthesis bound
1  Synthesis $u, \phi, G, R, b$
2      let $i \leftarrow$ size of $u$
3      for $k \leftarrow 0$ to $b$ do
4          let $A \leftarrow \{e$ in grammar G | e of size from $i - k$ to $i + k$ }
5          let $L \leftarrow \{\}$
6          forall $e \in A$ do
7              if $\phi[e/u]$ is valid then
8                  let $e.score \leftarrow \sum_{r_i \in R} r_i(e, u)$
9                  let $L \leftarrow L \cup e$
10             end
11         end
12         sort(L)                        ▷ by ascending order of score
13         if L is not empty then
14             return L.head              ▷ solution found
15         end
16     end
17     return FAIL
```

allow an efficient search procedure. Figure 5.3 describes a simplified grammar for SYNTH-LIB. Note that it allows the definition of integer expressions (*IntExpr*), including integer constants ($\mathcal{N}$), integer variables, and binary relations. Boolean expressions are defined similarly. Although simple, this grammar is sufficiently expressive for repairs over integers in booleans, including linear computations and logical relationships.

(2) Availability. SYNTH-LIB is not esoteric, but instead, broadly available to various tools for Syntax-Guided Synthesis (SyGuS) [4]. This allows for easy comparisons between tools, and indeed we use SYNTH-LIB to compare *S3* with two other state-of-the-art SyGuS solvers (*Enumerative* [4] and *CVC4* [77]). We believe that an abundance of synthesis techniques will benefit the program repair domain, given the rapid growth of the SyGuS research community, along with publicly available implementations [4, 77, 5].

(3) Cost Metrics. SYNTH-LIB allows for definition of cost metrics like expression size; this is useful for calculating ranking features. We further extended SYNTH-LIB to allow the specification of a starting *sketch*, which gives clues on where the enumeration procedure should start. In our case, the starting sketch is the original buggy expression, capturing our idea that the correct fix is more likely to be syntactically and semantically close to the original code. The sketch allows ranking features to measure the distance between candidate solutions and the original expression(s).

We illustrate with a SYNTH-LIB script for the example in Figure 5.1; Figure 5.4 shows the corresponding SYNTH-LIB script. In Figure 5.4, the first line sets the background theory of the language to Linear Integer Arithmetic (LIA). The function being synthesized f is of type INT→INT→BOOL→BOOL, (keyword $synth\text{-}fun$). The permitted solution space for the function f is described in its body, which allows expressions of type boolean. Each boolean expression can then be formed by logical relationships between any two integer or boolean expressions, via relational or logical operators. Expressions can also be variables; $M1$ in this case is a boolean expression. The allowed integer expression in the grammar is defined via $IntExpr$, which includes integer variables such as `charno` and $M2$, and constants such as 0.

We next define the constraints consisting of input-output examples and the starting sketch. Each constraint is defined by the keyword *constraint*. In our example, the first constraint says that if the value of $M2$ is 7, the value of $M1$ is `true`, and the value of $charno$ is 7, the expected output of the function f over `charno`, $M2$, and $M1$ is `true`. The second constraint can be interpreted similarly. These constraints corresponding to the extracted input-output examples described in Figure 5.2. A sketch, the starting-point expression, is defined by the keyword *sketch*. Here, the sketch is the original buggy expression u. Finally, the keyword *check-synth* instructs a synthesizer to start the synthesis process.

```
(set-logic LIA)
(synth-fun f ((charno Int) (M2 Int) (M1 Bool)) Bool
    ((Start Bool (
         (≤ IntExpr IntExpr) (< IntExpr IntExpr)
         (or Start Start) (and Start Start)
         M1 ))
     (IntExpr Int (
         charno M2 0
     ))))
(declare-var charno Int)
(declare-var M2 Int)
(declare-var M1 Bool)
(constraint (⇒ (and (= M2 7) (and (= M1 true) (= charno 7)))
            (= (f charno M2 M1) true)))
(constraint (⇒ (and (= M2 10) (and (= M1 true) (= charno 10)))
            (= (f charno M2 M1) true)))
(sketch u ((charno Int) (M2 Int) (M1 Bool)) Bool
            (and (and M1 (≤ 0 charno)) (< charno M2)))
(check-synth)
```

Figure 5.4: SYNTH-LIB script generated by *S3* for the example in Figure 5.1, derived using the "Alternatives" layer described in Figure 5.5. **M1** stands for `excerpt.equals(LINE)`, and **M2** stands for `sourceExcerpt.length()`.

5.3.2.2 Enumeration-based Synthesis.

S3 automatically generates a SYNTH-LIB script for each location under repair, and then uses an enumerative search to synthesize generalizable repair expressions conforming to the generated script. We note that multi-location repair can be achieved by generating the grammar for multiple functions simultaneously; we describe the process with respect to a single function for simplicity. We first explain how the SYNTH-LIB script is generated, and then the search procedure.

We divide the search space into multiple layers, each of which allows different components or operators, to appear in the SYNTH-LIB grammar script. If S3's search procedure cannot find a solution at a lower layer, it advances to the next. This approach tractably constrains the synthesis search space [68]. Figure 5.5 shows the six layers. The first layer allows *alternatives* of operators existing in the original buggy expression. For example, a pair {"&&", "||"} means that the operators in the pair are alternatives of one another. If the search procedure cannot find any solution, the grammar then cumulatively allows additional variables that do not exist in the original buggy expression,

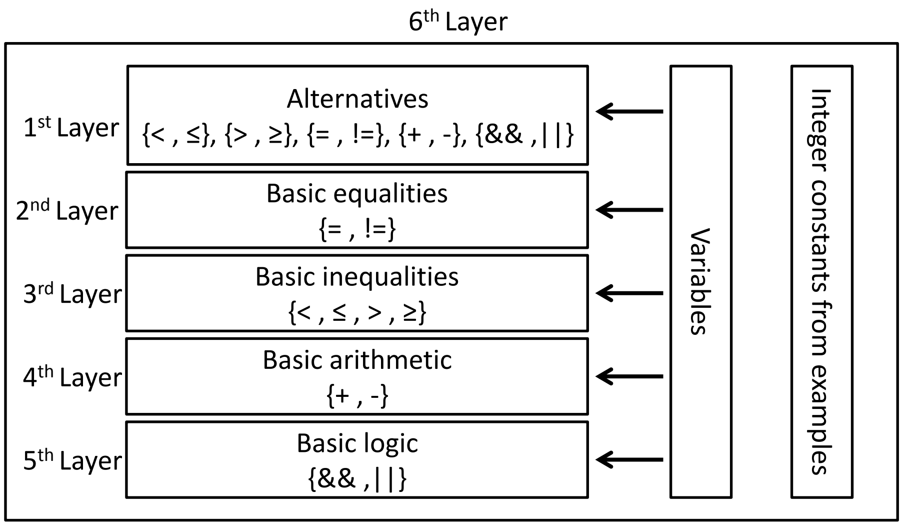


Figure 5.5: Search space layers specifiable in the grammar.

denoted by the "*Variables*" component in the Figure 5.5. At the second layer, the grammar allows *basic-inequalities* operators (= and *!*=), in addition to operators in the original expression. Again, if this search fails, it cumulatively allows for additional *Variables*. Subsequent layers can be interpreted similarly. We note that at the last (sixth) layer, the grammar allows all components, including integer constants appearing in the input-output examples. The reason integer constants are considered last is that such constants may unduly allow trivial solutions; this choice is influenced by previous studies [27, 49].

The design of separate sub-search-spaces systematically allows us to either prioritize which space to explore first, or unify the spaces freely. We heuristically prioritize the search space by automatically analyzing the surrounding context of the original buggy statement, such as the method declaration that contains the buggy statement. Particularly, S3 automatically looks for expressions in the surrounding context that use the same variables appearing in the buggy statement, and analyzes the components used in those expressions. This gives S3 clues on which search space to start from. If the prioritized search space does not help find solutions, S3 searches in the unified search space (the sixth layer). If S3 cannot find context to help prioritize the space, it follows the procedure described previously, starting from the first layer.

5.3.2.3 Ranking Features

Often, bug fixes (patches) can be quantified via the syntactic and/or semantic distances between the repaired and buggy programs. It has been proven that patches requiring minimal distances of both kinds are more likely correct [21]. Employing this insight, we thus propose features that measure the syntactic and semantic distance between a candidate solution and the original buggy code. The final ranking score of a candidate solution is the sum of individual feature scores. *S3* allows new features to be incorporated without difficulty; by contrast, constraint-based synthesis approaches (e.g. [66, 65]) typically require non-obvious Satisfiable Modulo Theory (SMT) encodings for new features [83].

Syntactic Features. Syntactic features look at differences between candidate solutions and the original buggy expression at the Abstract Syntax Tree (AST) level. We do this in three ways:

- **AST differencing**. We use GumTree [24] to compare ASTs. GumTree produces transformations between ASTs in the form of actions on AST nodes such as insert, delete, update, or move. We measure the number of actions needed to transform the original buggy AST to the candidate solution AST. This feature can be easily calculated by directly applying GumTree on the ASTs produced by parsing the SYNTH-LIB grammar script.
- **Cosine similarity**. An AST can also be represented as a vector of node occurrence counts [33]. The occurrence of each node type (e.g., integer variables or constants, or a binary operation) in an AST, represent a vector of the AST. The similarity of two ASTs can then be represented by the cosine similarity of their representative vectors, denoted as *cosine_score*. We then define the distance from the solution's AST to the original AST as: $1-$*cosine_score* (*cosine_score* of 1 denotes that two vectors are identical).

A SYNTH-LIB grammar explicitly enables type checking, meaning this feature is easy to calculate via an AST traversal to collect type information.

- **Locality of variables and constants**. Variables and constants are the primary ingredients of expressions. Thus, in addition to capturing abstract changes on the AST, we capture lower-level differences via the locations of variables and constants in expressions. We compute the Hamming distance between two vectors representing locations of variables and constants in each expression.[6] For example, consider $a \wedge (b < 1)$ as the original expression, $a \wedge (b \leq 1)$ as the first solution, and $(b \leq 1) \wedge a$ as the second solution. The hamming distance from the original expression for the first and second solutions are 0 and 3 respectively. Although both solutions are semantically equivalent, we may want to prefer the first in the interest of change minimality.

Semantic Features. Semantic features look at either the difference between a solution S_i and the original expression u, or the semantic quality of S_i itself. We propose three semantic features:

- **Model counting**. Model counting (c.f. [88]) is often used to count the number of models satisfying a particular formula. We use this feature to measure the level of "*disagreement*" between any two boolean expressions. That is, we say that a solution S_i and the original expression u disagree with each other if the formula $(S_i \wedge \neg u) \vee (\neg S_i \wedge u)$ is valid, meaning that S_i and u cannot be both valid at the same time. We then define the level of disagreement between S_i and u by the number of models that satisfy the formula, which accounts for the semantic distance between them. As a simple example, assume that we have: $a < 10$ as the original expression u, $a \leq 13$ as a solution S_1, and $a \leq 15$ as a solution S_2. The semantic distance via model counting between these solutions and u is 4 and 6, respectively. This simple example generalizes naturally to the typical off-by-one bug in

[6]`https://en.wikipedia.org/wiki/Hamming_distance`

Figure 5.1.

- **Output coverage**. This feature looks at how much a solution covers the set of outputs in the set of input-output examples. For instance, assume input-output examples (constraints) for two tests T_1 and T_2, on an input i, and an output o:

  T_1: $i = 5 \rightarrow o = 5$

  T_2: $(i = 6 \rightarrow o = 5) \vee (i = 6 \rightarrow o = 6)$

  A trivial solution for this example is simply the constant 5; Another solution is the expression i. The first solution overfits to only one output despite the presence of three examples that have two distinct outputs. The second solution covers all output scenarios in the provided examples, making it intuitively less overfitting as compared to the first. A solution S_i receives a O_i^{cov} score of N_c/N_o, where N_o is the number of output scenarios in the provided input-output examples, and N_c is the number of output scenarios that the solution S_i covers. The feature score of a solution S_i is defined as $1 - O_i^{cov}$. The higher O_i^{cov}, the better the solution S_i.

- **Anti-patterns**. This feature aims to heuristically prevent synthesis from generating trivial solutions. Particularly, these patterns are anti-duplicate and -constant expressions, e.g., $a < a$, $0 \neq 1$, etc. Expressions containing these patterns typically evaluate to a constant *true* or *false*, and are thus likely to overfit. We filter out these expressions during the synthesis process. Again, this can be easily done by traversing the AST produced by the SYNTH-LIB grammar. The utility of anti-patterns has been explored for search-based program repair [83], but not for semantics-based counterparts, partially because it is difficult to integrate additional such measures directly in the constraint-based synthesis approach [83].

5.4 Evaluation

This section describes our comparison between S3 and state-of-the-art semantics-based program repair techniques. We describe experimental setup and research questions in Section 5.4.1; answer those research questions in Sections 5.4.2–5.4.3; and present discussion, limitations, and threats in Section 5.4.4.

5.4.1 Experimental setup

We ran all experiments on a Intel Corei5 machine with 4 cores and 8GB of RAM.

Baseline approaches and settings We compare S3 to Angelix [66], Enumerative [4], and CVC4 [77]. Angelix offers its specification inference engine and synthesis engine in separate code packages. Although the specification inference engines behind Angelix and S3 work on C and Java programs, respectively, Angelix's synthesis engine takes as input example-based specifications like the synthesis engine of S3. Thus, to enable comparisons between S3 and Angelix, we instruct S3's inference engine to generate the same type of specifications that Angelix's synthesis engine uses, and instruct both S3's and Angelix's synthesis engines to synthesize the repair based on the same provided specifications. Enumerative [4] and CVC4 [77] are state-of-the-art Syntax-Guided Synthesis (SyGuS) engines which both take input in the form of SYNTH-LIB scripts, like S3.[7] This allows straightforward comparison between the tools.

For *single-line patches*, we run a repair synthesis tool on each buggy location of each program in parallel, and stop once a repair is found. The timeout for synthesis task is set to three minutes each. For *multi-line-patches*, we implement the approach described bellow.

[7]We refer interested readers to [2] and `http://www.sygus.org/` for a full comparison between SyGuS engines

Angelix tackles patches involving multiple lines [66] by grouping multiple buggy locations, and synthesizing repairs for several locations at once. Angelix clusters buggy locations into groups of a user-specified size by either locality or suspiciousness score produced by fault localization. We reimplemented this feature, following Angelix's source code.[8] Angelix's synthesis engine are run on these specifications.

We implemented our own strategy to tackle *multi-line patches* for S3, *Enumerative*, and *CVC4*. Each buggy location is repaired separately, after which patches for certain locations are grouped. Given a test suite T, and patches $\{P_i\}$ generated by a repair synthesis tool for location i. Assuming each patch $p \in P_i$ leads the program to pass a set of tests $T_i \subset T$, we iterate through all patches and combine those that have $\cup T_i = T$. The intuition is that combining these patches may render the whole test suite T to pass, which we then verify dynamically.

Datasets We consider two datasets of buggy programs:

- **Small programs associated with high coverage test suites.** We experiment with 52 Java bugs in the *smallest* subject programs of the IntroClass program repair benchmark [53] translated to Java [22]. The programs are student-written homework assignment from an introductory programming class; the goal of the programs is to find the smallest number between four integer numbers. Although the programs are small, they feature possibly complicated fixes involving changes in multiple *if-then-else* structures. We include only syntactically distinct programs. We focus on *smallest* because it only includes integer- and boolean-related fixes. Neither Angelix nor our framework can yet handle, e.g., floating point numbers or strings, primarily due to the limited capability of the constraint solving techniques used in symbolic execution.

[8]`https://github.com/mechtaev/angelix`. The implementation for this feature in Angelix's source is approximately 70 lines of Python code.

A key benefit of focusing on these small programs is that the problems in IntroClass are associated with two independent, high-quality test suites. We use one test suite to guide the search for a repair and the other to assess produced patch quality. We further augment the dataset by using Symbolic PathFinder [72] to generate additional tests. We do this by manually adding correctness specifications such as logical assertions, on the buggy programs, and use SPF to generate test inputs that expose bugs, e.g., assertion violations. This results in 16 additional tests.

- **Large real-world programs.** Our second dataset consists of 100 large real-world Java bugs from 62 subject programs, featuring ground truth bug fixes submitted by developers. Our dataset only includes bugs with patches that change fewer than five lines of code. This simplifies quality and correctness assessment of machine-generated patches, which is especially important because real-world test cases can be incomplete or weak specifications of desired behavior [75, 81].

  We build our dataset upon a previously-proposed bug fix history dataset [50], which originally consists of around 3000 likely bug-fixing commits of fewer than five lines of code collected from GitHub. To further ensure that the collected commits are actually bug fixes, we randomly sampled 500 commits, and manually checked them to ensure that the commits compile and that the program test cases expose bugs pre-commit (as compared to post-commit test behavior). We treat tests that fail in the before-patched version but pass in the patched version as the failing tests addressed by the bug fixing commit. Since this process is time consuming, we stopped once we found 100 bugs from 62 programs. Table 5.1 shows the top five largest programs for which S3 can correctly patch bugs. "KLoc" depicts the number of lines of Java code in each project.

Table 5.1: Top 5 largest programs that S3 can correctly patch. Math refers to the Apache Commons Math library

	Closure	**OrientDB**	**Math**	**Molgenis**	**Heritrix**
KLoc	237	203	175	54	48

Research questions and metrics. Our core metric is the number of buggy programs that a tool correctly patches. Fully assessing repair quality and correctness is an open problem in program repair research, and thus we approximate in several ways. For the IntroClass bugs, we designate a patch *correct* if it passes all held-out test cases, described above. We divide the SPF-generated tests randomly, using half to augment the tests used to repair and the other half to augment the held-out tests. For the real-world bugs, a patch is deemed *correct* if it is syntactically identical to the developer-produced patch. We also manually inspect all the results (produced by all repair tools) as a sanity check. In our inspection, if it is possible for a machine-generated patch to be converted into the corresponding developer's patch via basic transformations, we also consider it as correct. These patches are the minority in our evaluation; we separate these in our results and present the patches in prose. We report *overfitting rate*, or the percentage of produced patches that are incorrect, for each tool (lower is better); and *expressive power* in terms of the unique buggy programs each tool correctly patches. Our two research questions are then divided by dataset:

RQ1. How does each tool perform on the dataset of small programs associated with high coverage test cases, in terms of correct patches generated, overfitting rate, and expressive power?

RQ2. How does each tool perform on the dataset of real-world programs, in terms of correct patches generated, overfitting rate, and expressive power?

Table 5.2: Repair tool performance on 52 IntroClass bugs.

				Angelix			
	S3	**Enum**	**CVC4**	1	2	3	4
Produced	22	13	13	17	18	17	20
Pass all held-out tests	22	1	1	3	7	4	4
% Overfit	0%	92%	92%	82%	61%	76%	80%

```
if ((a < b) && (a < c) && (a < d)) {
    // By S3: ((a <= b) && (a <= c) && (a < d))
    // By Angelix: (a <= c) && (a < d)
    System.out.println(a);
} else if ((b < a) && (b < c) && (b < d)) {
    // By S3: (b <= a) && (b <= c) && (b < d)
    // By Angelix: no change
    System.out.println(b);
} else if ((c < a) && (c < b) && (c < d)) {
    // By S3: (c <= a) && (c <= b) && (c < d)
    // By Angelix: (c < d)
    System.out.println(c);
} else {
    System.out.println(d);
}
```

Figure 5.6: A bug in a *smallest* program correctly fixed exclusively by S3. We show the patches from S3 and Angelix.

5.4.2 Performance on IntroClass

Table 5.2 shows the results of each repair synthesis tool on 52 bugs from the IntroClass dataset. The "Produced" column shows the total number of patches that each tool generated that pass the provided test cases, while the "Pass held-out tests" shows the number of produced patches that generalize to pass all held-out evaluation tests (and that we thus consider correct). "% Overfit" shows the percentage of produced patches that do not generalize to the held-out tests (lower is better). Note that Angelix's multi-line patch facility is driven by two parameters: number of buggy locations in a group (1–4), and the criterion used to group them (either by locality or suspiciousness score). These results are based on score-based grouping, which uniformly outperformed the alternative in our experiments (results not shown). When the group size is set to 1, we allow Angelix to try our own multi-line patch strategy, in case single-line repair is unsuccessful.

Table 5.2 shows that S3 substantially outperforms the baselines, generating significantly more patches, *all* of which generalize to the held out test cases. The degree to which Angelix patches overfit varied by lines considered, ranging from a minimum of 61% to a maximum of 82%. Enumerative and CVC4 perform comparably, with a very high percentage of overfitting patches. S3 generates correct patches for all the bugs for which Angelix, Enumerative, and CVC4 can fix. S3 also generated almost exclusively multi-line patches (with one exception).

We speculate that the underlying synthesis techniques are the primary source of the baselines' weak performance. Enumerative enumerates expressions in increasing size, while CVC4 uses unsatisfiability (unsat) cores to synthesize solutions; neither rank candidate solutions, but instead conservatively return the first satisfying solution identified. Angelix encodes a simple patch minimality preference criteria in constraints suitable for PartialMax SMT. However, in these experiments, we observed that Angelix frequently generated patches that are quite different from the original buggy expressions (typically much smaller in size). These results and observations suggest that S3's combination of a customizable search space, an appropriately-managed expression-size-wise search strategy, and numerous ranking functions, all contribute to its successful generation of generalizable patches.

Figure 5.6 shows an example of a bug that S3 patches correctly but to which the baselines overfit. For brevity, we only show patches from S3 and Angelix. This code snippet requires a multi-line patch to multiple if-conditions. We show the replacement if-expressions from S3 and Angelix in the code comments. From the first if-condition, the Angelix fix is already incorrect, as it fails to capture the necessary relationship between variables a and b. The condition from S3 shares the structure of the original buggy expression, capturing the relationships between all variables. Producing this patch is likely assisted by S3's expression-size-wise enumerative search, which starts from the size of the original buggy expressions.

Table 5.3: Repair tool performance on 100 real-world bugs.

	S3	S3$_{syn}$	S3$_{sem}$	Enum	CVC4	Angelix
Produced	20	15	12	13	12	13
Syntax match	16	11	7	5	4	4
Manual	4	1	4	1	1	2
Overfit, Syn	20%	27%	42%	62%	67%	69%
Overfit, Both	0%	20%	8%	54%	58%	54%

```
...First bug...
- if (Character.isDigit(next)// Buggy if-condition
+ if (Character.isDigit(next) || next == '.') // fix by developer
+ if ((46 == next) || Character.isDigit(next)) // fix by S3

...Second bug...
- return (csvBuffer.getMark() >= (bufferIndex - 1))// fix buggy expression
+ return (bufferIndex) < (csvBuffer.getMark() + 1)// fix by developer
+ return (csvBuffer.getMark() > (bufferIndex - 1))// fix by S3

...Third bug...
- while (newLength > offset)// fix buggy expression
+ while (newLength < offset)// fix by developer
+ while (offset > newLength)// fix by S3

...Fourth bug...
if(this.runningState != STATE_RUNNING && this.runningState != STATE_SUSPENDED) {
    throw new IllegalStateException("...");
}
-        stopTime = System.currentTimeMillis();
+        if(this.runningState == STATE_RUNNING) { // fix by developer
+        if(this.runningState != STATE_SUSPENDED) // fix by S3
+            stopTime = System.currentTimeMillis();
+        }
```

Figure 5.7: Bugs for which S3 generates patches that are not syntactically identical but semantically equivalent to the developer fixes.

5.4.3 Performance on real-world programs

Table 5.3 shows the results of applying each considered repair tool on 100 real-world bugs from our second dataset. The first row shows the total number of bugs for which each tool generated a patch. Because we lack second independent test suites for these programs, we use a direct syntactic match to the developer patch to define correctness (row "Syntax match"). We additionally found, via manual inspection, a small number of additional patches that appear semantically identical to the developer patches; we describe these patches for S3 below. The last two rows show the percentage of produced patches that fail to generalize to capture the developer-written patch, as judged via strict syntactic match ("Overfit, Syn") or via both syntactic match and manual inspection ("Overfit, Both").

S3 again substantially outperforms the baseline techniques, generating correct patches for many more programs. Only 4 of the 20 S3 patches fail to strictly syntactically match the developer fixes. Although manual author inspection, is an inadequate mechanism for rigorously assessing patch quality, simple syntactic transformation rules can convert these patches to their developer equivalents; we separate these out in Figure 5.7.

In terms of overfitting, only 20% of S3's patches fail to generalize when judged by perfect syntactic fidelity; when manual inspection is considered, none of the patches overfit. For Angelix, Enumerative, and CVC4, 54%, 58%, and 54% of the produced patches overfit, respectively.

In these experiments, we also evaluate the relative contribution of S3's syntactic versus semantic feature sets for ranking — $\mathbf{S3_{syn}}$ and $\mathbf{S3_{sem}}$ in the table, respectively. When only either syntactic or semantic features are used to rank the solution space, the performances of S3 varies. $\mathbf{S3_{syn}}$ and $\mathbf{S3_{sem}}$ generate fewer correct patches, with slightly higher overfitting rates, suggesting that both kinds of features are beneficial for S3's performance. We additionally experimented with individual ranking feature of S3 as shown in Table 5.4. From the table, we can see that Cosine similarity and Locality of variables and constants are the most effective features, wherein each feature alone can fix 9 bugs. Model counting alone can fix 7 bugs, making it the least effective among the features.

Table 5.4: Effectiveness of individual ranking feature

Feature	#Bugs Fixed
Cosine Similarity	9
Anti-patterns	8
AST Differencing	8
Model Counting	7
Locality of variables and constants	9
Output Coverage	8

All programs that are correctly fixed by other tools are also fixed by S3. We note that the number of correctly-fixed bugs by the three baselines can be increased (to 9 bugs) if we combine all bugs correctly repaired by them. This combination is, however, still inferior to S3's performance.

The first bug in Figure 5.7 is an example of a bug that S3 fixes correctly, while the others do not. Enumerative and CVC4 generate the same fix with each other, that does not ultimately pass all tests (both synthesize `(0 == 0)` to replace the if condition); Angelix generates no fix for this bug. S3's fix is not syntactically identical but it is semantically equivalent to the developer's fix. This can be demonstrating by transforming S3's patch using basic transformation rules, e.g., swapping both left and right hand sides of the "||" operator, and converting the integer `46` to the character ".". The fix generated by Enumerative and CVC4, on the other hand, cannot be transformed to the developer's fix. We note that the incorrect fix generated by Enumerative and CVC4 is largely destructive, since it converts the branch condition to always evaluate to `true`. This kind of destructive fix can be prevented in S3 via the anti-patterns feature, as described in Section 5.3.2.3. In general, S3 generates more correct patches than the other approaches, judged via both syntactic fidelity to the developer fix and via fidelity with respect to basic syntactic transformations.

5.4.4 Discussion and Limitations

Semantics-based repair in general exclusively modifies expressions in conditions or on the right-hand side of assignments. Additionally, such techniques can only synthesize or reason about replacement code including boolean or integer types. Our experience suggests that these limitations are the primary reasons for unrepaired bugs in our experiments. Some bugs require large changes to semantic or control-flow structure (e.g., a change from `if(...){A};if(...){B}` to `if(...){A} else if(...){B}`), the insertion of new statements, or manipulation of variables of types that existing constraint solving technology cannot handle. Resolving these challenges remains future work, and can progress apace with progress in the synthesis domain. However, it is noteworthy that semantics-based repair techniques are reasonably expressive despite these limitations.

5.5 Conclusions

We proposed S3, a new repair synthesis system that is able to generate high-quality, general patches for bugs in real programs. S3 consists of two main phases, which serve to: (1) Automatically extract examples that serve as a specification of correct behavior, using dynamic symbolic execution on provided test cases, and (2) Use a synthesis procedure inspired by the programming-by-examples methodology to synthesize general patches. The efficiency and effectiveness of the synthesis procedure is enabled by our novel designs of three main parts, including a domain-specific language, which we extend from SYNTH-LIB [4]; an expression-size-wise enumerative search; and syntax- and semantic-guided ranking features that help rank the highest quality solutions highest in the solution space. Our results showed that S3 generates many more high-quality bug fixes than even the best performing baseline from prior work.

Beyond these results, our approach opens a number of opportunities for future repair synthesis techniques. The specifications, in the form of input-output examples, can be strengthened with specifications inferred by specification mining and other inference techniques [23, 43], possibly enabling integration of inductive and deductive synthesis for a more expressive overall system. Our dataset can also be extended, and used to evaluate many more repair systems. We plan to extend the SYNTH-LIB grammar to represent more tasks in the program repair domain, e.g., nonlinear computations on the integer domain. Finally, machine learning might be useful in automatically classifying bug types [85], to more effectively deal with different kinds of defects automatically.

Chapter 6

Reliability of Patch Correctness Assessment

The overfitting problem in APR is not only attributed to the way APR techniques generate and navigate the search space for repairs, but also the way generated patches are validated. In this chapter, we propose to assess the reliability of popular patch validation methodologies in the literature, namely automated annotation – in which an independent test suite is used to validate patches, and author annotation – in which authors of APR techniques validate patches generated by their and competing tools by themselves. We do this by first constructing a gold set of correctness labels for 189 randomly selected patches generated by 8 state-of-the-art APR techniques by means of a user study involving 35 professional developers as independent annotators. By measuring inter-rater agreement as a proxy for annotation quality – as commonly done in the literature – we demonstrate that our gold set is on par with other high-quality gold sets. Through an in-depth comparison of labels generated by author and annotated annotations and this gold set, we assess the reliability of the popular patch assessment methodologies. We subsequently report several findings and highlight their implications for future APR studies.

6.1 Introduction

Bug fixing is notoriously difficult, time-consuming, and costly [84, 12]. Hence, effective automatic program repair (APR) techniques that can help reduce the onerous burden of this task, is of tremendous value. Interest in APR has intensified as demonstrated by substantial recent work devoted to the area [65, 66, 95, 58, 60, 96, 52, 37, 48, 47, 45, 14], bringing the futuristic idea of APR closer to reality. APR can be generally divided into two main families including heuristics- vs semantics-based approaches, classified by the way they generate and traverse the search space for repairs.

Traditionally, test cases are used as the primary criteria for correctness judgment of machine-generated patches – a patch is deemed as *correct* if it passes all tests used for repair [52]. This assessment methodology, however, has been shown to be ineffective as there could be multiple patches passing all tests but are still indeed *incorrect* [74, 59]. Although the search space of ASR varies depending on the nature of underlying techniques, it is often huge and contains many plausible repairs, which unduly pass all tests but fail to generalize to the expected behaviours. This problem, which is often referred to as patch overfitting [81], motivates the need of new methodologies to assess patch correctness. The new methodologies need to rely on additional criteria instead of using the test suite used for generating repair candidates (aka. *repair test suite*) alone.

To address this pressing concern, most recent works have been following two methods for patch correctness assessment separately:

- **Automated annotation by independent test suite.** Independent test suites obtained via an automatic test case generation tool are used to determine correctness label of a patch – see for example [81, 49]. Following this method, a patch is deemed as *correct* or *generalizable* if it passes both the repair and independent test suites, and *incorrect* otherwise.

- **Author annotation.** Authors of ASR techniques manually check correctness labels of patches generated by their own and competing tools – see for example [95, 56]. Following this method, a patch is deemed as *correct* if authors perceive semantic equivalence between generated patches and original developer patches.

While the former is incomplete, in the sense that it fails to prove that a patch is actually correct, the latter is prone to author bias. In fact, these inherent disadvantages of the methods have caused an on-going debate as to which method is better for assessing the effectiveness of various APR techniques being proposed recently. Unfortunately, there has been no extensive study that objectively assesses the two patch validation methods and provides insights into how the evaluation of APR's effectiveness should be conducted in the future.

This study is conducted to address this gap in research. We start by creating a gold set of correctness labels for a collection of APR generated patches, and subsequently use it to assess reliability of labels created through author and automated annotations. We study a total of 189 patches generated by 8 popular APR techniques (ACS [95], Kali [74], GenProg [95], Nopol [96], S3 [46], Angelix [66], and Enumerative and CVC4 embedded in JFix [45]). These patches are for buggy versions of 13 real-world projects, of which six projects are from Defects4J [34] (Math, Lang, Chart, Closure, Mockito, and Time) and seven projects are from S3's dataset [46] (JFlex, Fyodor, Natty, Molgenis, RTree, SimpleFlatMapper, GraphHoper). To determine correctness of each patch, we follow best practice by involving multiple independent annotators in a user study. Our user study involves 35 professional developers; each APR-generated patch is labeled by five developers by comparing the patch with its corresponding ground truth patch created by the original developer(s) who fixed the bug. By analyzing the created gold set and comparing it with labels generated by three groups of APR tool authors [62, 56, 46] and two automatic test case generation tools such as DiffTGen [94] and Randoop [71], we seek to answer three research questions:

RQ1 *Can independent annotators agree on patch correctness?*

RQ2 *How reliable are patch correctness labels generated by author annotation?*

RQ3 *How reliable are patch correctness labels inferred through automatically generated independent test suite?*

In RQ1, by measuring inter-rater agreement as a proxy of annotation quality – as commonly done in the literature [16, 19] – we demonstrate that our gold set is on par with other high-quality gold sets. In the subsequent two RQs, we investigate the strengths and deficiencies of author and automated patch correctness annotation.

We summarize our contributions below:

- We are the first to investigate the reliability of author and automated annotation for assessing patch correctness. To perform such assessment, we have created a gold set of labelled patches created by a user study involving 35 professional developers. By means with this gold set, we highlight strengths and deficiencies of popular assessment methods employed by existing APR studies.

- Based on implications of our findings, we provide several recommendations for future APR studies to better deal with patch correctness validation. Especially, we find that automated annotation, despite being less effective as compared to author annotation, can be used to augment author annotation and reduce the cost of manual patch correctness assessment.

The rest of the chapter is organized as follows. We describe details of our user study to collect gold set of patch correctness labels in Section 6.2. Subsequently, we answer RQ1, RQ2, and RQ3 to assess the quality of our gold set, author annotation, and automated annotation in Section 6.3, 6.4, and 6.5

respectively. Section 6.6 discusses implications of our findings, our post-study survey, and threats to validity. We conclude and briefly describe future work in Section 6.7.

6.2 User Study

We conducted a user study with 35 professional developers to collect correctness labels of patches. In this study, every developer is required to complete several tasks by judging whether patches generated by APR tools are semantically equivalent to ground truth human patches.

Patch Dataset. Since the eventual goal of our study is to assess reliability of author and automated annotations, we need a set of patches that have been labeled before by APR tool authors and can be used as input to automated test case generation tools designed for program repair. We find the sets of patches recently released by Liu et al. [56], Martinez et al. [62], and Le et al. [46] to be suitable. Liu et al. and Martinez et al. label a set of 210 patches generated by APR tools designed by their research groups (i.e., ACS [95], and Nopol [96]) and their competitors (i.e., GenProg [52], Kali [74]). Le et al. label a set of 79 patches generated by their APR tool (i.e., S3 [46]) and their competitors (i.e., Angelix [66], and Enumerative and CVC4 embedded in JFix [45]). The authors label these patches by manually comparing them with ground truth patches obtained from version control systems of the corresponding buggy subject programs.[1] These patches can be used as input to DiffTGen, which is a state-of-the-art test generation tool specifically designed to evaluate patch correctness [94], and Randoop – a popular general purpose test case generation tool [71].

Due to resource constraint, i.e., only 35 professional developers agree to spend an hour of their time in this user study, we cut down the dataset to

[1]Since authors of [56] and [95] overlap, we can use the labels to evaluate reliability of author labelling.

Table 6.1: Selected Patches and their Author Label

	GenProg	**Kali**	**Nopol**	**ACS**	**S3**	**Angelix**	**Enum**	**CVC4**
Incorrect	14	14	84	4	0	7	6	6
Correct	4	1	6	14	10	2	4	4
Unknown	2	2	5	0	0	0	0	0
Total	20	17	95	18	10	9	10	10

189 patches by randomly selecting these patches from their original datasets. Details of the dataset of 189 patches are shown in Table 6.1.

Task Design. At the start of the experiment, every participant is required to read a tutorial that briefly explains automated program repair and what they need to do to complete the tasks. Afterwards, they can complete the tasks one-by-one through a web interface.

Figure 6.2 shows the screenshot of an example task that we give to our user study participants through a web interface. For each task, we provide a ground truth patch taken from the version control system of the corresponding buggy subject program, along with a patch that is generated by an automated program repair tool. We also provide additional resources including full source code files that are repaired by the patch, link to the GitHub repository of the project, outputs of failing test cases[2], and source code of the failing test cases. Based on this information, participants are asked to evaluate the correctness of the patch by answering the question: *Is the generated patch semantically equivalent to the correct patch?* To answer this question, participants can choose one of the following options: "Yes", "No" or "I don't know". Finally, if they wish to, they can provide some reasons that explain their decision. Our web interface will record participants' answers and the amount of time they need to complete each task.

Participants and Task Assignment. Thirty three of the 35 professional developers participating in this study work for two large software development companies (named Company C1 and C2), while another two work as engineers

[2]These information is generated using Defects4J [34] info command.

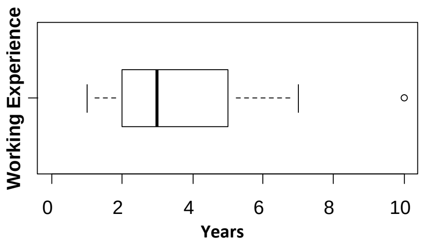


Figure 6.1: Distribution of participant work experience

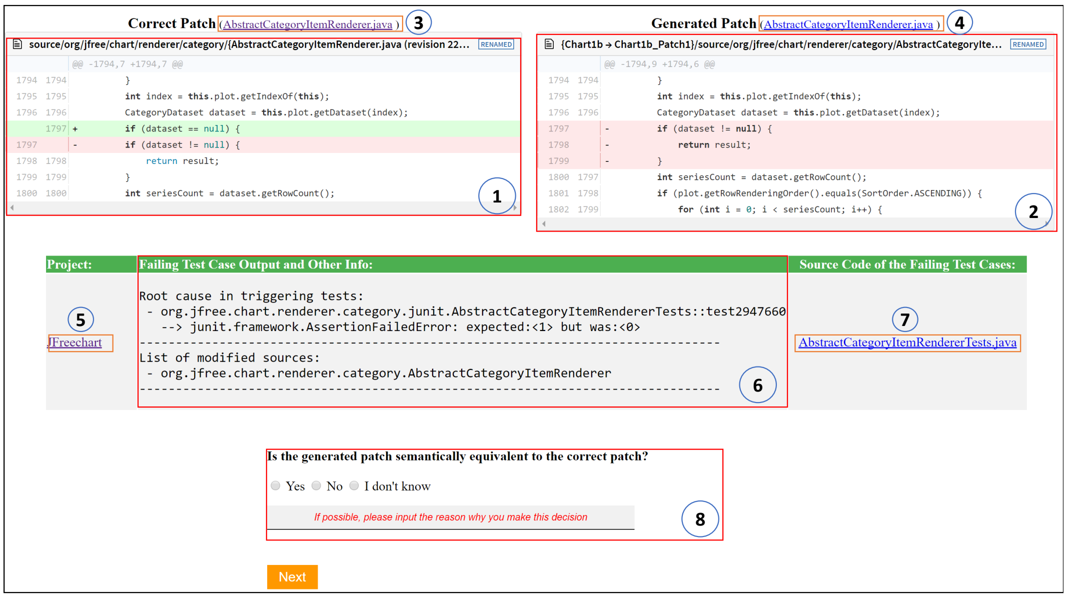
for an educational institution. Company C1 currently has more than 500 employees and Company C2 has more than 2000 employees. Both companies have a large number of active projects that expose developers to various business knowledge and software engineering techniques. All the 35 developers work for projects that use Java as the main programming language.

Figure 6.1 shows the distribution of years of work experience of our participants. The average number of years of work experience that these participants have is 3.5. Two developers from the educational institution are very senior, who have worked for 5.5 and 10 years, respectively. The most experienced developer from industry has worked for seven years, while some has only worked for one year. Based on their working experience, we group participants into two groups: *junior* and *senior*. There are 20 *junior* developers and 15 *senior* developers, respectively.

We divided the 35 participants into seven groups. The ratio of *junior* and *senior* developers for each group was kept approximately the same. Each patch generated by program repair tools is labeled by five participants. Participants in the same group receive the same set of patches to label.

6.3 Assessing Independent Annotators' Labels

Our user study presented in Section 6.2 was conducted to build a set of gold standard labels for machine-generated patches, which can reliably be used to assess reliability of author and automated annotations. Before using the labels produced by our user study, we need to first ascertain their quality. Agreement


Correct Patch (AbstractCategoryItemRenderer.java)
3
source/org/jfree/chart/renderer/category/{AbstractCategoryItemRenderer.java (revision 22…
RENAMED
1
Generated Patch (AbstractCategoryItemRenderer.java)
4
{Chart1b → Chart1b_Patch1}/source/org/jfree/chart/renderer/category/AbstractCategoryIte…
RENAMED
2
Project:
5
JFreechart
Failing Test Case Output and Other Info:
6
Source Code of the Failing Test Cases:
7
AbstractCategoryItemRendererTests.java
8
Next

```
@@ -1794,7 +1794,7 @@
            }
            int index = this.plot.getIndexOf(this);
            CategoryDataset dataset = this.plot.getDataset(index);
+           if (dataset == null) {
-           if (dataset != null) {
                return result;
            }
            int seriesCount = dataset.getRowCount();
```

```
@@ -1794,9 +1794,6 @@
            }
            int index = this.plot.getIndexOf(this);
            CategoryDataset dataset = this.plot.getDataset(index);
-           if (dataset != null) {
-               return result;
-           }
            int seriesCount = dataset.getRowCount();
            if (plot.getRowRenderingOrder().equals(SortOrder.ASCENDING)) {
                for (int i = 0; i < seriesCount; i++) {
```

```
Root cause in triggering tests:
 - org.jfree.chart.renderer.category.junit.AbstractCategoryItemRendererTests::test2947660
   --> junit.framework.AssertionFailedError: expected:<1> but was:<0>
--------------------------------------------------------------------------------
List of modified sources:
 - org.jfree.chart.renderer.category.AbstractCategoryItemRenderer
--------------------------------------------------------------------------------
```

Is the generated patch semantically equivalent to the correct patch?

Yes No I don't know

If possible, please input the reason why you make this decision

Figure 6.2: A sample task viewed through our web interface. (1) and (2) are the correct patch and the patch generated by an ASR tool; (3) and (4) are the links to source code files that contain the patches; (5) is the link to the corresponding project's GitHub repository; (6) and (7) are the output of the failed test cases and their source files; (8) is the question we asked a participant to answer.

among annotators is often used as a measure of quality [16, 20, 79]. Thus, in this section, we investigate the degree to which the annotators agree with one another. This answers RQ1: Can independent annotators agree on patch correctness?

Methodology. To answer RQ1, we first compute some simple statistics highlighting the number of agreements and disagreements among annotators. We then calculate several well-accepted measures of inter-rater reliability. Finally, we perform some sanity checks to substantiate whether or not annotators are arbitrary in making their decisions.

Results. To recap, our annotators are 35 professional developers who are tasked to annotate 189 machine-generated patches. Each patch is annotated by five professional developers; each provides either one of the following labels: incorrect, correct, or unknown. Table 6.2 summarizes the number of agreements and disagreements among annotators. The number of patches in which all developers agree on each patch's label is 118 (62.4% of all patches); of which

Table 6.2: Results of participant annotations. First column indicates the number of patches that every developer agrees on the label of each patch as correct or incorrect. Second column indicates the number of patches, wherein each patch has least one developer labeling it as unknown and the remaining developers agrees on the label of the patch. Last column indicates the number of patches that the label of each patch can be determined by a majority voting among developers' labels.

	All Agree	**All Agree - Unk**	**Majority Agree**
Incorrect	95	132	152
Correct	23	23	35
Total	118	155	187

95 patches are labeled as incorrect and 23 patches are labeled as correct. Moreover, ignoring unknown labels, the number of patches for which the remaining annotators fully agree on their labels is 155 (82.0% of all patches). Out of these, the numbers of patches that are labeled as incorrect and correct are 132 and 23, respectively. Lastly, for 187 out of 189 patches (98.9% of all patches), there is a majority decision (i.e., most annotators agree on one label). Out of these, 152 and 35 patches are identified as incorrect and correct, respectively.

We also compute several inter-rater reliability scores: mean pairwise Cohen's kappa [16, 18] and Krippendorff's alpha [41]. Using the earlier test we consider three different ratings (i.e., correct, incorrect, and unknown), while the latter test allows us to ignore unknown ratings[3]. Inter-rater reliability scores measure how much homogeneity, or consensus, there is between raters/labelers. The importance of rater reliability hinges on the fact that it represents the extent to which the data collected in the study are correct representations of the variables being measured. A low inter-rater reliability suggests that either the rating scale used in the study is defective or raters need to be retrained for the rating task or the task is highly subjective. The higher the inter-rater reliability the more reliable the data is.

Table 6.3 shows details of interpretations of reliability score values by Landis and Koch [42]. It is worth noting that there is another interpretation of

[3]Krippendorff's alpha allows us to have different number of ratings for each data point.

Table 6.3: Interpretation of Inter-Rater Reliability Scores by Landis and Koch [42].

Score Range	**Interpretation**
< 0	poor agreement
$[0.01, 0.20]$	slight agreement
$[0.21, 0.40]$	fair agreement
$[0.41, 0.60]$	moderate agreement
$[0.61, 0.80]$	substantial agreement
$[0.81, 1.00]$	almost perfect agreement

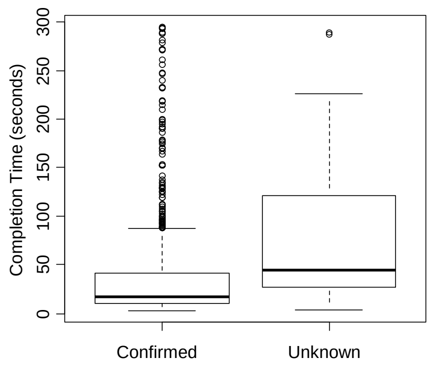


Figure 6.3: Time taken by annotators to decide whether a patch's label is either known (confirmed as correct or incorrect) or unknown.

kappa value by Manning *et al.* [16], which indicates that a kappa value falling between 0.67 and 0.8 demonstrates a fair agreement between raters – the second highest level of agreement by their interpretation. It has been shown that this fair level of inter-rater agreement normally happens in popular datasets such as those used for TREC evaluations[4] and medical IR collections [16].

The computed mean pairwise Cohen's kappa and Krippendorff's alpha for our data are 0.691 and 0.734 respectively, which highlight a substantial agreement among participants and satisfies the standard normally met by quality benchmark datasets.

To further validate the annotations, we perform two sanity checks to substantiate whether or not annotators are arbitrary in their decisions:

- First, we expect conscientious annotators to spend more time inspecting patches that are eventually labeled as unknown than other patches. An-

[4]Text REtrieval Conference (TREC), which is championed by US National Institute of Standards and Technology (NIST) since 1992, provides benchmark datasets for various text retrieval tasks – see `http://trec.nist.gov/data.html`.

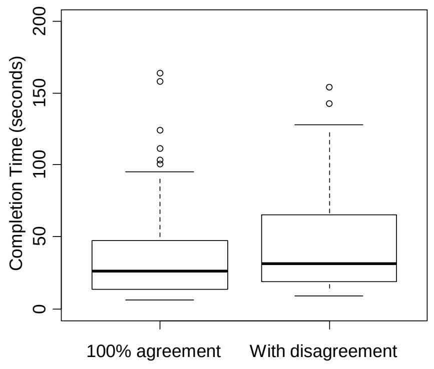


Figure 6.4: Time taken by annotators to decide a patch's label for full-agreement and disagreement cases.

notators who label patches as unknown without thinking much would be likely making arbitrary decisions. Figure 6.3 depicts a box plot showing the time participants took on patches that are labeled as unknown and other patches. It can be seen that participants took more time on the earlier set of patches. Wilcoxon signed-rank test returns a p-value that is less than 0.005, indicating a statistically significant difference. Moreover, the Cliff's delta[5], which is a non-parametric effect size measure, is 0.469 (medium).

- Second, we expect conscientious annotators to spend more time inspecting difficult patches than easy ones. We consider disagreement among annotators as proxy for patch difficulty. We compare the time taken by participants in identifying patches for which there is complete agreement to those for which disagreement exists. Figure 6.4 shows a box plot which shows that participants spend more time on disagreement cases. Wilcoxon signed-rank test returns a p-value that is less than 0.05, indicating statistically significant difference. Moreover, the Cliff's delta is 0.178 (small).

The above results substantiate the quality of our dataset. In the subsequent sections, which answer RQ2 and RQ3, we use two versions of our dataset ALL-AGREE (see "All Agree" column in Table 6.2) and MAJORITY-AGREE (see "Majority Agree" column in Table 6.2), to assess the reliability of author and automated annotations.

[5]Cliff defines a delta of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large effect size, respectively [17].

Table 6.4: Results of labels by authors compared to independent annotators.

	Indep Annotators-Authors	**All Agree**	**Majority Agree**
Same	Incorrect-Incorrect	82	133
	Correct-Correct	23	33
Different	Incorrect-Correct	6	10
	Correct-Incorrect	0	2
	Incorrect-Unknown	7	9
	Correct-Unknown	0	0
	Total	118	187

6.4 Assessing Author Annotation

A number of studies proposing automated repair approaches evaluate the proposed approaches through manual annotation performed by authors [56, 95, 50]. Author subjectivity may cause bias which can be a threat to the internal validity of the study. Author bias has been actively discussed especially in the medical domain, e.g., [87]. Unfortunately so far, there has been no study that investigates presence or absence of bias in author annotation and its impact to the validity of the labels in automated program repair. This section describes our effort to fill this need by answering RQ2: How reliable is author annotation?

Methodology. Recall that our user study makes use of patches released by three research groups, including Liu et al. [56], Martinez et al. [62], and Le et al. [46] who created program repair tools namely ACS, Nopol, and S3, respectively. Authors of each tool manually labeled the patches generated by their tool and its competing approaches by themselves. To answer RQ2, we compare labels produced by the three research groups with those produced by our independent annotators whose quality we have validated in Section 6.3. We consider the ALL-AGREE and MAJORITY-AGREE datasets mentioned in Section 6.3.

Results. Table 6.4 shows the detailed results on the comparisons between authors' labels and independent annotators' labels. We found that for ALL-AGREE dataset, authors' labels match with independent annotators' labels

```
 1  @@ -115,9 +115,7 @@ public class StopWatch {
 2  public void stop() {
 3     if(this.runningState != STATE_RUNNING && this.runningState != STATE_SUSPENDED) {
 4        throw new IllegalStateException("...");
 5     }
 6  +  if(this.runningState == STATE_RUNNING)// Developer patch
 7  +  if(-1 == stopTime)// Generated patch
 8        stopTime = System.currentTimeMillis();
 9     this.runningState = STATE_STOPPED;
10  }
```

Figure 6.5: An example of a patch that has mismatched labels. Liu et al. identified the patch (shown at line 7) as correct, while independent annotators identified this patch as incorrect. The ground truth (developer) patch is shown at line 6.

(Same) for 105 out of 118 patches (89.0%). There are 13 patches for which authors' labels mismatch those by independent annotators (Different). Among these patches, 6 are identified by independent annotators as incorrect, but identified by authors as correct (Incorrect-Correct). For the other 7 patches, authors' labels are unknown while independent annotators' labels are incorrect (Incorrect-Unknown). For the MAJORITY-AGREE dataset, 88.8% of the labels match. There are 21 mismatches; 10 belong to Incorrect-Correct cases, 2 to Correct-Incorrect cases, and 9 to Incorrect-Unknown cases. Figure 6.5 shows an example patch generated by Nopol [96] that has mismatched labels. It is labeled as correct by Martinez et al. and incorrect by independent annotators.

We also compute inter-rater reliability of authors' labels and labels in ALL-AGREE and MAJORITY-AGREE datasets. The Cohen's kappa values are 0.719 and 0.697 considering the ALL-AGREE and MAJORITY-AGREE datasets, respectively[6]. Comparing these scores with Landis and Koch's interpretation in Table 6.3, there is substantial agreement.

A majority (88.8-89.0%) of patch correctness labels produced by author annotation match those produced by independent annotators. Inter-rater reliability scores indicate a substantial agreement between author and independent annotator labels.

To better characterize cases where author and independent annotator la-

[6]The Krippendorf's alpha values are 0.717 and 0.695

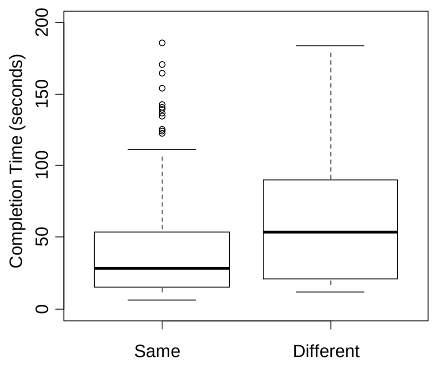


Figure 6.6: Participant completion time for patches for which author and independent annotator labels match (Same) and those whose labels mismatch (Different)

bels match (Same) and those where they do not match (Different), we investigate the time that participants of our user study took to label the two sets of patches. Since the number of mismatches is smaller in the ALL-AGREE dataset, we focus on comparing labels in MAJORITY-AGREE dataset. Figure 6.6 depicts a box plot showing the distribution of completion time corresponding to the two sets of patches. According to the figure, patches with matching labels took participants a shorter period of time to label comparing to those whose labels mismatched. Wilcoxon signed-rank test returns a p-value that is less than 0.05, indicating statistically significant difference. The Cliff's delta is equal to 0.278 (small). Since task completion time can be used as a proxy for measuring task difficulty or lack thereof [92], we consider participants completion time as a proxy of difficulty in assessing patch correctness. The result suggests that disagreements between authors and independent annotators happen for more difficult cases.

6.5 Assessing Automated Annotation

In this research question, we investigate the reliability of the use of automatically generated independent test suite (ITS) in annotating patch labels. ITS has been used as an objective proxy to measure patch correctness – a patch is deemed as incorrect if it does not pass the ITS, and as correct or generalizable otherwise [81, 49]. It is unequivocal that incorrect patches determined by ITS are indeed incorrect. However, it is unclear if ITS can detect a large proportion of incorrect patches. Moreover, the extent to whether correct (generalizable) patches determined by ITS are indeed correct remain questionable. Thus, to assess the usefulness of ITS, we investigate the answer to RQ3: How reliable is automatically generated ITS in determining patch correctness?

Methodology: We employ the recently proposed test case generation tool DiffTGen by Xin *et. al* [94] and Randoop [71] to generate ITS. To generate ITS using DiffTGen and Randoop, the human-patched program is used as ground truth. For DiffTGen, we run using its best configuration reported in [94], allowing it to invoke Evosuite [25] in 30 trials with the search time of each trial limited to 60 seconds. A machine-generated patch is identified as *incorrect* if there is a test in the DiffTGen-generated ITS that witnesses the output differences between the machine and human patches. For Randoop, we run it on the ground truth program with 30 different seeds with each run limited to 5 minutes. A machine-generated patch is identified as incorrect if there is at least one test case in the Randoop-generated ITS that exhibits different test results in machine-patched and human-patched (ground truth) programs, e.g., it fails on the machine-patched program but passes on the ground truth program, or otherwise. By this way, we allow both tools to generate multiple test suites. It is, however, worth noting that DiffTGen and Randoop are incomplete in the sense that they do not guarantee to always generate the test cases that witness incorrect patches.

We use test cases generated by the tools to automatically annotate the 189 patches and compare the generated labels to those in ALL-AGREE and MAJORITY-AGREE datasets which are created by our user study.

Results: Out of the 189 patches in our study, DiffTGen generates test cases that witness 27 incorrect (overfitting) patches. Details of these patches are shown in Table 6.6. The ALL-AGREE ground truth identifies 17 of these 27 patches as incorrect (the other 10 patches lie outside of the ALL-AGREE

Table 6.5: Kappa values when using DiffTGen, Randoop, and their combination to label patches in ALL-AGREE and MAJORITY-AGREE datasets.

	All Agree			**Majority Agree**		
	DiffT	Rand	Comb	DiffT	Rand	Comb
Cohen's Kappa	0.078	0.073	0.158	0.075	0.072	0.146
Kripp's Alpha	-0.32	-0.3	-0.057	-0.336	-0.313	-0.097

dataset), while the MAJORITY-AGREE dataset identifies all of them as incorrect. Unfortunately, most of the patches labelled as incorrect in ALL-AGREE (65 patches) and MAJORITY-AGREE (121 patches) datasets failed to be detected as such by ITS generated by DiffTGen. Randoop performs similarly as compared to DiffTGen. It identifies 31 patches as incorrect, all of which are also identified as incorrect in the MAJORITY-AGREE dataset. Note that, DiffTGen and Randoop when combined can identify totally 51 unique patches as incorrect.

In their studies, Smith et al. [81] and Le et al. [81] assume a patch is incorrect if it does not pass an ITS, and correct or generalizable otherwise. Using the same assumption to generate correctness labels, we can compute inter-rater reliability between labels automatically annotated by running ITS generated by DiffTGen and Randoop and labels in ALL-AGREE and MAJORITY-AGREE datasets. As readers may have expected, the kappa values are very low as shown in Table 6.5, e.g., Cohen's kappa values when using DiffTGen-generated ITS for ALL-AGREE and MAJORITY-AGREE are 0.078 and 0.075, repsectively.[7]

> Independent test suite generated by DiffTGen and Randoop can only label fewer than a fifth of incorrect patches as such in ALL-AGREE and MAJORITY-AGREE datasets.

We now compare author labels discussed in Section 6.4 with ITS labels. Table 6.6 shows the author labels of the 27 and 31 patches identified as incorrect by DiffTGen and Randoop, respectively. For these patches, the majority of

[7]The corresponding Krippendorff's alpha values are -0.32 and -0.336

the labels by authors and DiffTGen match. However, there are three special patches identified as incorrect by DiffTGen, including Math_80 generated by Kali, Chart_3 generated by GenProg, and Math_80_2015 generated by Nopol, while author labels are “Unknown”. One special patch identified as incorrect by Randoop (Math_73 generated by GenProg), is labelled as correct by authors.

Table 6.6: Labels by Independent annotators (“Annot” column) and authors (“Authors” column) of patches identified by independent test suite (ITS) generated by DiffTGen or Randoop as incorrect .

		DiffTGen	**Randoop**	**Annot**	**Authors**
Kali	Time_4	Incorrect	Incorrect	Incorrect	Incorrect
	Math_32	Incorrect		Incorrect	Incorrect
	Math_2	Incorrect		Incorrect	Incorrect
	Math_80	Incorrect		Incorrect	Unknown
	Math_95	Incorrect	Incorrect	Incorrect	Incorrect
	Math_40	Incorrect		Incorrect	Incorrect
	Chart_13	Incorrect		Incorrect	Incorrect
	Chart_26	Incorrect		Incorrect	Incorrect
	Chart_15	Incorrect	Incorrect	Incorrect	Incorrect
	Chart_5	Incorrect	Incorrect	Incorrect	Incorrect
GenProg	Math_2	Incorrect		Incorrect	Incorrect
	Math_8	Incorrect		Incorrect	Incorrect
	Math_80	Incorrect		Incorrect	Incorrect
	Math_81	Incorrect		Incorrect	Incorrect
	Math_95	Incorrect	Incorrect	Incorrect	Incorrect
	Math_40	Incorrect		Incorrect	Incorrect
	Math_73		Incorrect	Incorrect	Correct
	Chart_1		Incorrect	Incorrect	Incorrect
	Chart_3	Incorrect		Incorrect	Unknown
	Chart_5	Incorrect	Incorrect	Incorrect	Incorrect
	Chart_15	Incorrect	Incorrect	Incorrect	Incorrect
Nopol	Math_33	Incorrect		Incorrect	Incorrect
	Math_73_2017		Incorrect	Incorrect	Incorrect
	Math_80_2017	Incorrect		Incorrect	Incorrect
	Math_80_2015	Incorrect		Incorrect	Unknown
	Math_97	Incorrect		Incorrect	Incorrect
	Math_105		Incorrect	Incorrect	Incorrect
	Time_16		Incorrect	Incorrect	Incorrect
	Time_18		Incorrect	Incorrect	Incorrect
	Chart_13_2017	Incorrect		Incorrect	Incorrect
	Chart_13_2015	Incorrect		Incorrect	Incorrect
	Chart_21_2017	Incorrect		Incorrect	Incorrect
	Chart_21_2015	Incorrect		Incorrect	Incorrect
	Closure_7		Incorrect	Incorrect	Incorrect
	Closure_12		Incorrect	Incorrect	Incorrect
	Closure_14		Incorrect	Incorrect	Incorrect
	Closure_20		Incorrect	Incorrect	Incorrect
	Closure_30		Incorrect	Incorrect	Incorrect
	Closure_33		Incorrect	Incorrect	Incorrect
	Closure_76		Incorrect	Incorrect	Incorrect
	Closure_111		Incorrect	Incorrect	Incorrect
	Closure_115		Incorrect	Incorrect	Incorrect
	Closure_116		Incorrect	Incorrect	Incorrect
	Closure_120		Incorrect	Incorrect	Incorrect
	Closure_124		Incorrect	Incorrect	Incorrect
	Closure_130		Incorrect	Incorrect	Incorrect
	Closure_121		Incorrect	Incorrect	Incorrect
	Mockito_38		Incorrect	Incorrect	Incorrect
Angelix	Lang_30		Incorrect	Incorrect	Incorrect
CVC4	Lang_30		Incorrect	Incorrect	Incorrect
Enum	Lang_30		Incorrect	Incorrect	Incorrect

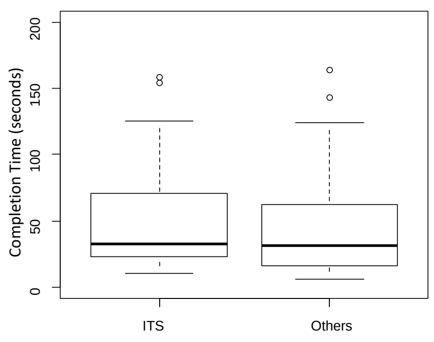


Figure 6.7: Participant completion time for the 51 unique patches labelled by DiffTGen's and Randoop's ITSs as incorrect versus that for other patches.

Finally, we want to investigate the difficulty of judging correctness of patches that DiffTGen and Randoop generated ITSs label as incorrect. To do so, we compare participant completion time for the set of 51 unique patches and the set of other patches. Figure 6.7 shows time spent by participants labelling these two sets of patches. We find that they are more or less the same. Wilcoxon signed-rank test confirms that the difference is not statistically significant. Thus, patches that ITS successfully label as incorrect are not necessarily the ones that participants require more time to manually label.

6.6 Discussion

In this section, we first provide implications of our findings. We then discuss our post-study survey, in which we asked a number of independent annotators for rationales behind their patch correctness judgements.

6.6.1 Implications

To recap, we have gained insights into the reliability of patch correctness assessment by authors and by automatically generated independent test suite (ITS); each of them has their own advantages and disadvantages. Based on these insights, we provide several implications as follows.

Authors' evaluation of patch correctness should be made publicly available to the community.

Liu et al., Martinez et al., and Le et al. released their patch correctness labels publicly [56, 62, 46], which we are grateful for. We believe that considerable effort has been made by authors to ensure the quality of the labels. Still, we notice that for slightly more than 10% of the patches, authors' labels are different from the ones produced by multiple independent annotators. Thus, we encourage future APR paper authors to release their datasets for public inspection. The public (including independent annotators) can then provide inputs on the labels and possibly update labels that may have been incorrectly assigned. Our findings here (e.g., author annotations are fairly reliable) may not generalize to patches labelled by authors which have not been released publicly. It is possible that the quality of correctness labels for those patches (which are not made publicly available) to be lower. Also, as criticized by Monperrus *et al.* [67], the conclusiveness of the evaluation of techniques that keep patches and their correctness labels private is questionable.

Collaborative effort is needed to distribute the expensive cost of APR evaluation.

In this study, we have evaluated correctness of 189 automatically generated patches by involving independent annotators. We have shown that the quality of the resultant labels (measured using inter-rater reliability) are on par with high-quality text retrieval benchmarks [16]. Unfortunately, evaluation using independent annotators is expensive. To evaluate 189 patches, we need to get 35 professional developers; Each agrees to spend up to an hour of their time. This process may not be scalable especially considering the large number of new APR techniques that are released in the literature year by year. Thus, there is a need for a more collaborative effort to distribute the cost of APR evaluation. One possibility is to organize a competition involving impartial

```
@@ -168,7 +168,7 @@ public class PearsonsCorrelation {
        } else {
            double r = correlationMatrix.getEntry(i, j);
            double t = Math.abs(r * Math.sqrt((nObs - 2)/(1 - r * r)));
+           out[i][j] = 2 * tDistribution.cumulativeProbability(-t);
-           out[i][j] = 2 * (1 - tDistribution.cumulativeProbability(t));
        }
```

(a) Human Patch

```
@@ -190,6 +190,7 @@
        for (int j = 0; j < i; j++) {
          double corr = correlation(matrix.getColumn(i), matrix.getColumn(j));
          outMatrix.setEntry(i, j, corr);
+         if(1 - nVars < -1)
          outMatrix.setEntry(j, i, corr);
        }
```

(b) Generated Patch

Figure 6.8: A machine-generated patch labeled by ITS as incorrect but labeled by author annotation as unknown.

industrial data owners (e.g., software development houses willing to share some of their closed bugs) who are willing to judge correctness of generated patches. Similar competitions with industrial data owners have been held to advance various fields such as forecasting[8] and fraud detection[9].

> Independent test suite (ITS) *alone* should not be used to evaluate the effectiveness of APR.

Independent test suites (ITSs) generated by DiffTGen [94] and Randoop [71] have been shown to be ineffective in annotating correctness labels for patches (see Section 6.5). Only fewer than a fifth of the incorrect patches are identified as such by ITSs generated by DiffTGen and Randoop. Based on effectiveness of state-of-the-art test generation tool for automatic repair that we assessed in this study, we believe that ITS *alone* should not be used for *fully automated* patch labeling. The subject of ITS generation for program repair is new though and we encourage future studies to improve the quality of automatic test generation tools so that more incorrect patches can be detected. That being said, automated patch annotation may not be a silver bullet; the general problem of patch correctness assessment (judging the equiv-

[8]http://www.cikm2017.org/CIKM_AnalytiCup_task1.html
[9]http://research.larc.smu.edu.sg/fdma2012/

alence of developer patch and automatically generated patch) is a variant of program equivalence problem which has been proven to be undecidable with no algorithmic solution [80].

Independent test suite, despite being less effective, can be used to augment author annotation.

It has been shown in Section 6.5 that ITS generated by DiffTGen and Randoop identified four patches as incorrect whereas the labels generated by author annotation are unknown and correct. An example of such patch is shown in Figure 6.8. From the figure, we can notice that it is hard to judge whether the patch is correct or incorrect. From this finding, we believe that ITS, despite being less effective than author annotation in identifying correct patches, can be used to augment author annotation by helping to resolve at least some of the ambiguous cases. Authors can possibly run DiffTGen and Randoop to identify clear cases of incorrect patches; the remaining cases can then be manually judged. The use of both author and automated annotation via ITS generation can more closely approximate multiple independent annotators' labels while requiring less cost.

6.6.2 Post-Study Survey

We conducted a post-study survey to investigate why a developer chooses a different answer from the majority. Among the 189 patches, there are several patches where the majority, but not all participants, agree on patch correctness. Among participants annotating these patches, we selected 11 who answered differently from the majority and emailed them to get deeper insights into their judgments. In our email, we provided a link to the same web interface used in our user study to allow participants to revisit their decision for the patch in question. Notice that we did not inform the participants that their answers were different from the majority. We received replies from 8 out of the 11 participants (72.7% response rate).

We found that 5 out of 8 developers changed their correctness labels after they looked into the patch again; their revised labels thus became consistent with the labels that the majority agree. The remaining three kept their correctness labels; two judged two different patches as incorrect (while the majority labels are correct) while another judged a patch as correct (while the majority label is incorrect). These participants kept their decision for different reasons; one was unsure of a complex expression involved in the patch, another highlighted a minor difference that may be considered ignorable by others, and the other participant viewed the generated and ground truth patch to have similar intentions. An excerpt of the patch in question for the last mentioned participant is shown in Figure 6.9.

```
@@ -83,7 +83,7 @@ public class StringEscapeUtils {
         public static String escapeJava(String str) {
+                return escapeJavaStyleString(str, false, false);
-                return escapeJavaStyleString(str, false);
        }
                              ......
@@ -242,9 +241,7 @@ public class StringEscapeUtils {
                                out.write('\\');
                                break;
                            case '/' :
+                               if (escapeForwardSlash) {
                                    out.write('\\');
+                               }
                                out.write('/');
                                break;
                            default;
```

(a) Human Patch

```
@@ -239,6 +239,7 @@
                            case '\\' :
                                out.write('\\');
                                out.write('\\');
+                                  if(escapeSingleQuote)
                                break;
                            case '/' :
                                out.write('\\');
```

(b) Generated Patch

Figure 6.9: An example of a patch in post-study

6.7 Conclusion and Future Work

In this chapter, to assess reliability of existing patch correctness assessment methods, we conducted a user study with 35 professional developers to construct a gold set of correctness labels for 189 patches generated by different APR techniques. By measuring inter-rater agreement (which was found to be substantial and on par with other high-quality benchmarks), we validated the quality of annotation labels in our gold set. We then compare our gold set with labels produced by authors (i.e., Liu et al. [56], Martinez et al. [62], and Le et al. [46]) and independent test suites generated by DiffTGen [94] and Randoop [71], and report their strengths and deficiencies. In particular, we find that a majority (88.8-89.0%) of patch correctness labels generated by authors match those produced by independent annotators. On the other hand, only fewer than a fifth of incorrect patches can be labelled by independent test suites (ITSs) generated by DiffTGen and Randoop as such. DiffTGen and Randoop can however generate ITSs that can uncover multiple incorrect patches that are labeled as "unknown" or "correct" by authors. Based on our findings, we recommend that APR authors release their patch correctness labels for public inspection. We also encourage more collaborative effort to distribute the expensive cost of APR evaluation especially through user studies like ours. We also stressed that ITS alone should not be used to fully judge patch correctness labels; still, they can be used in conjunction with author annotation to help the latter produce labels that can more closely approximate independent annotators' labels.

In the future, we plan to expand our gold set by recruiting more professional developers and collecting more patches generated by additional APR techniques through a large-scale collaborative effort among APR researchers. We also plan to explore the possibility of organizing competitions with industrial data owners (e.g., with our two industrial partners whose developers have participated in this study) for further APR research.

Chapter 7

Dissertation's Conclusion & Future Plans

Motivation of this dissertation: Bug fixing is time-consuming and costly. Hence, automated program repair (APR) techniques that can relieve the burden on human developers in bug fixing would be of tremendous value. Substantial recent works have been proposed to automatically repair variety of bugs in many real-world large software, gradually materializing the futuristic idea of APR. These APR techniques, despite varying in the ways they search for repairs, commonly rely on test cases to guide the repair process and validate machine-generated patches. The reliance on test cases is, in fact, problematic to research in APR since test cases are known to be incomplete, in a sense that they often insufficiently encode desired behaviors of software. This could lead APR techniques to generate patches that overfit to the test cases used for repair, but do not necessarily generalize to expected behavior that developers would expect. To overcome the mentioned problem – often regarded as patch overfitting, APR techniques must address the followings: (1) maintaining both scalability and tractability, in which APR techniques must cheaply scale to large, real-world programs, while being able to tackle the large search space for repairs for those programs to find correct repairs, (2) enhancing expressive power to correctly fix many more real bugs from diverse real-world programs (3) methodologies to validate machine-generated patches.

Accomplishment of this dissertation: This dissertation tackles the above challenges posed by the overfitting problem by (1) proposing new search- and semantics-based APR techniques that are capable of generating *generalizable* repairs, (2) empirically studying the overfitting issue in semantics-based APR, complementing existing study on the search-based counterparts, and (3) empirically evaluating the reliability of patch validation methodologies, providing insightful guidelines on how machine-generated patches should be evaluated. In particular, we proposed *HDRepair* – a search-based APR technique that leverages the development history of many software to guide and drive the repair process. We empirically study various characteristics of different semantics-based APR techniques, showing that APR techniques are indeed subject to overfitting at various degrees. We subsequently proposed *S3* – a semantics-based APR technique that systematically constrains the syntactic search space for repairs and effectively ranks solutions to find correct repairs. Finally, we study the reliability of existing popular patch validation methodologies, and provide several guidelines and insights on how APR-generated patches should be evaluated.

Threats to validity: This dissertation used a few benchmarks to perform evaluations of program repair tools, including IntroClass [53], Defects4J [34], and S3's dataset [46]. These benchmarks contain either small programs (IntroClass), or small number of real bugs (fewer than 200 bugs). This poses threats to the generalizability of findings reported in this dissertation.

Future work: There are several future directions for this dissertation, including (1) more detailed study on characterizing overfitting behaviors of semantics-based repair, (2) consideration of developer intentions via partial specifications, and (3) creation of datasets that better benefit the APR community. First, a further study on why whitebox tests are worse/better than blackbox tests in helping APR mitigate overfitting would be useful. It would help characterize which tests are more important for APR, suggesting potentially better ways for APR to traverse the search space when using tests, e.g., a patch passing more tests that are important could be more likely to be correct. Second, developers often express their intentions via partial specifications, e.g., assertions. A repair technique that takes these intentions into consideration to generate generalizable repairs could be of tremendous value. Third, datasets specialized for APR are needed for better assessment of APR techniques. For example, a dataset with a large number of bugs (e.g., thousands of bugs), covering various bug patterns that happen in practice would help systematically characterize strengths and weaknesses of different APR techniques.

Other future directions include a plan to improve both search- and semantics-based APR further to better tackle the overfitting problem. It would also be interesting to find ways to leverage the best of both APR families, e.g., merging search- and semantics-based APR to be more expressive. Also, applications of modern machine learning techniques into APR to leverage the large amount of historical data existing in public code repositories would also be interesting.

Bibliography

[1] Github archive. `https://githubarchive.org/`.

[2] Syntax-guided synthesis. `http://www.sygus.org/`, 2016.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[4] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering*, 2015.

[5] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. Technical report, University of Pennsylvania.

[6] John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Workshop on Eclipse Technology eXchange*, pages 35–39, San Diego, California, 2005.

[7] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 1–10, Honolulu, HI, USA, 2011.

[8] Algirdas Avizienis, J-C Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004.

[9] James E Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the second international conference on genetic algorithms*, pages 14–21, 1987.

[10] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[11] Christian Bird, Adrian Bachmann, Eirik Aune, John Duffy, Abraham Bernstein, Vladimir Filkov, and Premkumar Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 121–130, Amsterdam, The Netherlands, August 2009.

[12] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. Technical report, University of Cambridge, Judge Business School, 2013.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, 2008.

[14] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *International Conference on Software Engineering*, ICSE'11, pages 121–130, 2011.

[15] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in System Dependability (HotDep)*, number DSLAB-CONF-2009-002, 2009.

[16] D Manning Christopher, Raghavan Prabhakar, and Schütze Hinrich. Introduction to information retrieval. *An Introduction To Information Retrieval*, 151:177, 2008.

[17] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114(3):494, 1993.

[18] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[19] Tadele T Damessie, Thao P Nghiem, Falk Scholer, and J Shane Culpepper. Gauging the quality of relevance assessments using inter-rater agreement. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1089–1092. ACM, 2017.

[20] Tadele Tedla Damessie, Thao P. Nghiem, Falk Scholer, and J. Shane Culpepper. Gauging the quality of relevance assessments using inter-rater agreement. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval, Shinjuku, Tokyo, Japan, August 7-11, 2017*, pages 1089–1092, 2017.

[21] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification (CAV)*, pages 383–401. Springer, 2016.

[22] Thomas Durieux and Martin Monperrus. IntroClassJava: A benchmark of 297 small and buggy java programs. Technical report, Universite Lille 1, 2016.

[23] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[24] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *International Conference on Automated Software Engineering*, ASE'14, pages 313–324, 2014.

[25] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 416–419, 2011.

[26] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–187, 2012.

[27] Sumit Gulwani, Javier Esparza, Orna Grumberg, and Salomon Sickert. Programming by examples (and its applications in data wrangling). *Verification and Synthesis of Correct and Secure Systems*, 2016.

[28] Mark Harman. The current state and future of search based software engineering. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 342–357, 2007.

[29] Mark Harman. Automated patching techniques: the fix is in: technical perspective. *Communications of the ACM*, 53(5):108–108, 2010.

[30] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Working Conference on Mining Software Repositories (MSR)*, pages 121–130, 2013.

[31] Pieter Hooimeijer and Westley Weimer. Modeling bug report quality. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 34–43. ACM, 2007.

[32] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *International Conference on Software Engineering (ICSE)*, pages 215–224, Cape Town, South Africa, 2010.

[33] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *International conference on Software Engineering (ICSE)*, pages 96–105. IEEE, 2007.

[34] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *International Symposium on Software Testing and Analysis*, ISSTA '14, pages 437–440, 2014.

[35] David Kawrykow and Martin P Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 351–360. ACM, 2011.

[36] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. Repairing programs with semantic code search. In *International Conference on Automated Software Engineering (ASE)*, pages 295–306, 2015.

[37] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering*, ICSE '13, pages 802–811, 2013.

[38] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design*, FMCAD, pages 91–100. IEEE, 2011.

[39] Robert Könighofer and Roderick Bloem. Repair with on-the-fly program analysis. In *Haifa Verification Conference*, pages 56–71. Springer, 2012.

[40] John R. Koza. Genetic programming: On the programming of computers by means of natural selection, 1992. *See http://miriad. Iip6. fr/microbes Modeling Adaptive Multi-Agent Systems Inspired by Developmental Biology*, 229, 1992.

[41] Klaus Krippendorff. Estimating the reliability, systematic error, and random error of interval data. *Educational and Psychological Measurement*, 30(1):61–70, 1970.

[42] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[43] Tien-Duy B Le, Xuan-Bach D Le, David Lo, and Ivan Beschastnikh. Synergizing specification miners through model fissions and fusions (t). In *International Conference on Automated Software Engineering (ASE)*, pages 115–125. IEEE, 2015.

[44] Xuan-Bach D Le. Towards efficient and effective automatic program repair. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 876–879. IEEE, 2016.

[45] Xuan Bach D Le, Duc Hiep Chu, David Lo, and Claire Le Goues. Jfix: Sematics-based repair of Java programs via Symbolic Pathfinder. In *Proceedings of the 2017 ACM International Symposium on Software Testing and Analysis*, ISSTA, 2017.

[46] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: syntax- and semantic-guided repair synthesis via programming

by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 593–604, 2017.

[47] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. Enhancing automated program repair with deductive verification. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 428–432. IEEE, 2016.

[48] Xuan-Bach D Le, Tien-Duy B Le, and David Lo. Should fixing these failures be delegated to automated program repair? In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 427–437, 2015.

[49] Xuan-Bach D Le, David Lo, and Claire Le Goues. Empirical study on synthesis engines for semantics-based program repair. In *International Conference on Software Maintenance and Evolution*, ICSME'16, pages 423–427, 2016.

[50] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE, 2016.

[51] Xuan-Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. Overfitting in semantics-based automated program repair. *Empirical Software Engineering, under minor revision*, 2017.

[52] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering*, ICSE'12, pages 3–13, 2012.

[53] Claire Le Goues, Neal Holtschulte, Edward K Smith, Yuriy Brun, Premkumar Devanbu, Stephanie Forrest, and Westley Weimer. The ManyBugs

and IntroClass benchmarks for automated repair of C programs. *Transactions on Software Engineering (TSE)*, 41(12):1236–1256, Dec. 2015.

[54] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[55] Ben Liblit, Alex Aiken, Alice X Zheng, and Michael I Jordan. Bug isolation via remote program sampling. In *ACM Sigplan Notices*. ACM, 2003.

[56] Xinyuan Liu, Muhan Zeng, Yingfei Xiong, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. *arXiv preprint arXiv:1706.09120*, 2017.

[57] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. *SIGPLAN Not.*, 47(10):133–146, October 2012.

[58] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *European Software Engineering Conference and International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, 2015.

[59] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *International Conference on Software Engineering (ICSE)*, pages 702–713. ACM, 2016.

[60] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Symposium on Principles of Programming Languages (POPL)*, pages 298–312, 2016.

[61] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, pages 1–29, 2016.

[62] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering*, 22(4):1936–1964, 2017.

[63] Matias Martinez and Martin Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[64] Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 492–495. ACM, 2014.

[65] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, pages 448–458. IEEE Press, 2015.

[66] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701. IEEE, 2016.

[67] Martin Monperrus. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242. ACM, 2014.

[68] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *International*

Conference on Software Engineering (ICSE), pages 772–781. IEEE Press, 2013.

[69] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[70] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. The class-level mutants of mujava. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 78–84, New York, NY, USA, 2006. ACM.

[71] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 75–84, 2007.

[72] Corina S Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Automated Software Engineering*, 20(3):391–425, 2013.

[73] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[74] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis*, pages 24–36. ACM, 2015.

[75] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch gen-

eration systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 24–36. ACM, 2015.

[76] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[77] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *International Conference on Computer Aided Verification (CAV)*, pages 198–216, 2015.

[78] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark Barrett. Counterexample-guided quantifier instantiation for synthesis in smt. In *CAV*, 2015.

[79] Falk Scholer, Andrew Turpin, and Mark Sanderson. Quantifying test collection quality based on the consistency of relevance judgements. In *Proceeding of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2011, Beijing, China, July 25-29, 2011*, pages 1063–1072, 2011.

[80] Michael Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.

[81] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[82] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. Codeflaws: A programming competition benchmark for evaluating automated program repair tools. In *ICSE Poster*, 2017. To appear.

[83] Shin Hwei Tan, Hiroaki Yoshida, Mukul R Prasad, and Abhik Roychoudhury. Anti-patterns in search-based program repair. In *International Symposium on Foundations of Software Engineering*, pages 727–738. ACM, 2016.

[84] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *Planning Report, NIST*, 2002.

[85] Ferdian Thung, Xuan-Bach D Le, and David Lo. Active semi-supervised defect categorization. In *International Conference on Program Comprehension*, pages 60–70. IEEE Press, 2015.

[86] Jacobo Torán. On the hardness of graph isomorphism. *SIAM J. Comput.*, 33(5):1093–1108, May 2004.

[87] Alexander R. Vaccaro, Alpesh Patel, and Charles Fisher. Author conflict and bias in research: Quantifying the downgrade in methodology. *Spine*, 30(14), 2011.

[88] Willem Visser. What makes killing a mutant hard. In *International Conference on Automated Software Engineering (ASE)*, pages 39–44. ACM, 2016.

[89] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[90] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 356–366. IEEE Press, 2013.

[91] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the*

Fourth International Workshop on Mining Software Repositories, page 1. IEEE Computer Society, 2007.

[92] Christopher D Wickens. Processing resources and attention. *Multiple-task performance*, 1991:3–34, 1991.

[93] Leigh Williamson. Ibm rational software analyzer: Beyond source code. In *Rational Software Developer Conference. http://www-07. ibm. com/in/events/rsdc2008/presentation2. html*, 2008.

[94] Qi Xin and Steven P Reiss. Identifying test-suite-overfitted patches through test case generation. In *International Symposium on Software Testing and Analysis*, pages 226–236. ACM, 2017.

[95] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. Precise condition synthesis for program repair. In *International Conference on Software Engineering*, pages 416–426. IEEE Press, 2017.

[96] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *Transactions on Software Engineering*, 2016.

[97] Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 286–295. ACM, 2003.