



Efficient Exploration and Analysis of Program Repair Search Spaces

KHASHAYAR ETEMADI SOMEOLIAYI

Doctoral Thesis
Stockholm, Sweden, 2024

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Division of Theoretical Computer Science
TRITA-EECS-AVL-2024:76
978-91-8106-071-3
SE-10044 Stockholm
Sweden

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av Teknologie doktorexamen i datalogi måndagen den 22 november 2024 klockan 10.00 i Kollegiesalen, Brinellvägen 6, Kungliga Tekniska Högskolan, Stockholm.

© Khashayar Etemadi Someoliayi, October 3, 2024

Tryck: Universitetsservice US AB

Abstract

The ubiquitous presence of software has made its failures a huge source of cost. To avoid this cost, fixing buggy programs is essential. As manual debugging is notoriously resource demanding, researchers have proposed to replace it with automated program repair (APR) approaches. As input, an APR approach takes a specification and a buggy program that violates the specification; it then tries to generate a patched program that complies with the specification. APR works in two major phases: search space exploration, and search space analysis. In the search space exploration phase, the APR approach predicts the potentially faulty parts of the code (fault localization) and replaces them with various alternatives (patch generation). In this phase, finding the best alternatives can be highly costly, which makes APR inefficient. In the search space analysis phase of APR, the patches explored in the previous phase are analyzed to identify a correct patch. This phase consists of three components: 1) requires validating the patches against the given specification (patch validation); 2) automatically detecting patches more likely to be correct (automated patch assessment); and 3) manually identifying a correct patch (manual patch assessment). The automated patch assessment should be lightweight and accurate to make APR efficient by minimizing the number of manual assessment required. The manual patch assessment is also a deciding factor in APR efficiency as it needs human expertise, the most valuable development resource. To ensure APR is practical and can replace manual debugging, we have to make it efficient.

In this thesis, we aim at improving the efficiency of both exploration and analysis of APR search spaces. For this, we particularly target the efficiency of patch generation, automated patch assessment, and manual patch assessment. This thesis makes five contributions as follows.

We make two contributions to make patch generation efficient. First, we introduce SORALD, a template-based APR approach for fixing SONARJAVA static warnings. SORALD employs accurately designed templates to generate exactly one patch that is highly likely to fix the bug. The lightweight patch generation technique and the small search space that needs little analysis makes SORALD an efficient APR approach. Our second contribution is CIGAR, an iterative LLM-based APR tool that fixes test-suite failing bugs. CIGAR uses three delicately designed prompts and two prompting strategies to a large number of likely correct patches with minimal LLM token cost. Most notably, CIGAR uses execution-based feedback to the LLM to guide the model toward generating likely correct patches without repeating itself.

We also make two other contributions to make automated patch assessment efficient. First, we propose LIGHTER, a lightweight tool for estimating the potential of fix templates used by template-based APR approaches. LIGHTER compares fix templates against historical bug-fixes by developers to assess if the templates follow the modification patterns used by experts. The result of this assessment is used to rank the patches based on the potential of templates used for their generation. Our second contribution to APR automated patch assessment is MOKAV. MOKAV is an execution-driven iterative

LLM-based tool that generates tests to differentiate between patches. The generated tests help grouping APR patches into separate clusters and manually analyzing only one patch from each cluster. This makes APR efficient by significantly reducing the manual patch assessment effort.

Our last contribution is COLLECTOR-SAHAB, which aims at helping code reviewers better understand behavioral changes caused by patches. Given two versions of a program P & Q, COLLECTOR-SAHAB collects the execution trace of both P & Q. It next compares the traces and identifies runtime differences at variable and return value level. Finally, it augments the code diff between P & Q with a concise selection of extracted runtime differences. This code augmentation helps code reviewers to better understand the behavior of APR patches and thus, reduces human effort needed for manual patch assessment.

Keywords: Automated Program Repair, Program Repair Efficiency, Large Language Models

Sammanfattning

Den allestädes närvarande närvaron av programvara har gjort dess misslyckanden till en enorm kostnadskälla. För att undvika denna kostnad är det viktigt att fixa buggyprogram. Eftersom manuell felsökning är notoriskt resurskrävande, har forskare föreslagit att det ska ersättas med tillvägagångssätt för automatiserad programreparation (APR). Som input tar en APR-metod en specifikation och ett buggyprogram som bryter mot specifikationen; den försöker sedan generera ett korrigerat program som överensstämmer med specifikationen. APR fungerar i två huvudfaser: sökumsutforskning och sökum-sanalys. I sökutrymmesutforskningsfasen förutsäger APR-metoden de potentiellt felaktiga delarna av koden (fellokalisering) och ersätter dem med olika alternativ (patchgenerering). I denna fas kan det vara mycket kostsamt att hitta de bästa alternativen, vilket gör den effektiva röntan ineffektiv. I sökutrymmesanalysfasen av APR analyseras lapparna som utforskades i föregående fas för att identifiera en korrekt lapp. Denna fas består av tre komponenter: 1) kräver validering av patcharna mot den givna specifikationen (patchvalidering); 2) automatisk detektering av plåster med större sannolikhet att vara korrekta (automatisk plåsterbedömning); och 3) manuell identifiering av en korrekt patch (manuell patchbedömning). Den automatiska lappbedömningen bör vara lätt och exakt för att göra APR effektiv genom att minimera antalet manuella bedömningar som krävs. Den manuella patchbedömningen är också en avgörande faktor för effektiv APR eftersom den behöver mänsklig expertis, den mest värdefulla utvecklingsresursen. För att säkerställa att APR är praktiskt och kan ersätta manuell felsökning, måste vi göra det effektivt.

I den här avhandlingen syftar vi till att förbättra effektiviteten i både utforskning och analys av APR-sökutrymmen. För detta fokuserar vi särskilt på effektiviteten av patchgenerering, automatiserad patchbedömning och manuell patchbedömning. Denna avhandling ger fem bidrag enligt följande.

Vi gör två bidrag för att göra patchgenerering effektiv. Först introducerar vi SORALD, en mallbaserad APR-metod för att fixa SONARJAVA statiska varningar. SORALD använder noggrant utformade mallar för att generera exakt en patch som med stor sannolikhet kommer att fixa buggen. Den lätta patchgenereringstekniken och det lilla sökutrymmet som kräver lite analys gör SORALD till en effektiv APR-metod. Vårt andra bidrag är CIGAR, ett iterativt LLM-baserat APR-verktyg som fixar fel i testpaketet. CIGAR använder tre noggrant designade uppmaningar och två uppmaningsstrategier för ett stort antal sannolikt korrekta patchar med minimal LLM-tokenkostnad. Mest anmärkningsvärt är att CIGAR använder exekveringsbaserad feedback till LLM för att vägleda modellen mot att generera sannolikt korrekta patchar utan att upprepa sig.

Vi gör också två andra bidrag för att göra automatiserad patchbedömning effektiv. Först föreslår vi LIGHTER, ett lättviktigt verktyg för att uppskatta potentialen för fixmallar som används av mallbaserade APR-metoder. LIGHTER jämför fixmallar med historiska buggfixar från utvecklare för att bedöma om mallarna följer modifieringsmönster som används av experter. Resultatet av denna bedömning används för att rangordna patcharna baserat på potentialen hos mallar som används för deras generering. Vårt andra bidrag till

APR automated patch assessment är MOKAV. MOKAV är ett exekveringsdrivet iterativt LLM-baserat verktyg som genererar tester för att skilja mellan patchar. De genererade testen hjälper till att gruppera APR-lappar i separata kluster och manuellt analysera endast en patch från varje kluster. Detta gör APR effektiv genom att avsevärt minska den manuella patchbedömningen.

Vårt sista bidrag är COLLECTOR-SAHAB, som syftar till att hjälpa kodgranskare att bättre förstå beteendeförändringar orsakade av patchar. Givet två versioner av ett program P & Q, samlar COLLECTOR-SAHAB in exekveringsspår för både P & Q. Därefter jämförs spåren och identifierar körtidsskillnader på variabel- och returvärdesnivå. Slutligen utökar den koddifferensen mellan P & Q med ett kortfattat urval av extraherade körtidsskillnader. Denna kodförstärkning hjälper kodgranskare att bättre förstå beteendet hos APR-lappar och minskar på så sätt den mänskliga ansträngningen som krävs för manuell patchbedömning.

*I dedicate this work to Professor Martin Monperrus, a great man and a perfect
scientist*

List of Papers

The list of papers that are included in the thesis in chronological order:

I *Estimating the potential of program repair search spaces with commit analysis*

Khashayar Etemadi, Niloofar Tarighat, Siddharth Yadav, Matias Martinez, and Martin Monperrus
in Journal of Systems and Software (2022)

Author Contribution: The author of this thesis designed LIGHTER, participated in its implementation, designed and performed the experiments and wrote the paper. Niloofar Tarighat and Siddharth Yadav participated in implementing LIGHTER. Martin Monperrus supervised the project and participated in writing the paper.

II *Sorald: Automatic patch suggestions for sonarqube static analysis violations*

Khashayar Etemadi, Nicolas Harrand, Simon Larsen, Haris Adzemovic, Henry Luong Phu, Ashutosh Verma, Fernanda Madeiral, Douglas Wikstrom and Martin Monperrus
in IEEE Transactions on Dependable and Secure Computing (2022)

Author Contribution: The author of this thesis participated in the design and implementation of SORALD. He also designed and participated in conducting the experiments. He was also the main author of the paper. Nicolas Harrand participated in conducting the experiments. Simon Larsen participated in implementing SORALD. Martin Monperrus supervised the project. All authors participated in reviewing and writing the paper.

III *Augmenting diffs with runtime information*

Khashayar Etemadi, Aman Sharma, Fernanda Madeiral and Martin Monperrus
in IEEE Transactions on Software Engineering (2023)

Author Contribution: The author of this thesis designed COLLECTOR-SAHAB, participated in its implementation, designed and participated in conducting the experiments and was the main author of the paper. Aman Sharma participated in implementing COLLECTOR-SAHAB. All authors contributed to writing the paper. Martin Monperrus supervised the project.

IV *Cigar: Cost-efficient program repair with llms*

David Hidvegi, **Khashayar Etemadi**, Sofia Bobadilla and Martin Monperrus

under review in IEEE Transactions on Software Engineering

Author Contribution: The author of this thesis participated in the design and implementation of CIGAR, wrote the paper, participated in designing and conducting the experiments. David Hidvegi was the main person in charge of implementing CIGAR and participated in the design and conducting the experiments. All authors participated in reviewing and writing the paper. Martin Monperrus supervised this project.

V *Mokav: Execution-driven Differential Testing with LLMs*

Khashayar Etemadi, Bardia Mohammadi, Zhendong Su and Martin Monperrus

under review in Journal of Systems and Software

Author Contribution: The author of this thesis participated in designing and implementing MOKAV and designing and conducting the experiments. He was the main author of the paper as well. Bardia Mohammadi participated in the design of MOKAV and was the main person in charge of implementing it. All the authors participated in reviewing and writing the paper. Martin Monperrus and Zhendong Su supervised this project.

Acknowledgement

I must thank Aman, Andre, Deepika, Monica, Karolina, XX for letting me win the publish and perish game on that night in Sep, 2024. I owe you everything :) Thank you all for making the right decision and voting for who truly deserved to win that game. Sofia, you arrived late and missed the chance to vote for the winner, but I know you would do the right thing if you were present. Thanks a lot!

The list is long, I also want to thank all other past and present ASSERT members who made my PhD an awesome experience that I will remember forever.

Fernanda, I have already told you how much I appreciate your help over email. I will not forget your help. Benoit, thanks for everything you did for me, especially, for reminding me that I had to write a thesis in July, when I was on a terribly-timed vacation.

Martin, you are great scientist and a perfect supervisor. If you were not this good, I would never be able to earn a PhD. Luckily, I was smart enough to understand what a great researcher and person you are, the first time we had an interview. That made me choose to do my PhD at KTH with you, what a flawless decision! I respect science and hate cheating it; this made you an ideal supervisor for me. Thank you for your inspiration!

The reader might wonder why I have not mentioned other people who are important in my personal life. That is because I find a PhD thesis too mediocre and boring to talk about them. They are beyond this.

This five-year period was a special experience for me; I will leave a part of my heart here in Stockholm. There is much more to say, but as that Persian guy from the middle ages complains, this notebooks comes to an end and the story remains. So, I end this here and leave the rest to the astute observer!

Acronyms

List of commonly used acronyms:

APR	Automated program repair
AST	Abstract syntax tree
LLM	Large language model
NMT	Neural machine translation
OOV	Out-of-vocabulary

Contents

List of Papers	vii
Acknowledgement	ix
Acronyms	xi
Contents	1
1 Introduction	3
1.1 Automated Program Repair	4
1.2 Problem Statement	8
1.3 Thesis Contributions	10
1.4 Thesis Outline	18
2 State of the Art	19
2.1 Automated Program Repair Workflow	19
2.2 Patch Generation Techniques	21
2.3 Automated Patch Assessment	33
2.4 Manual Patch Assessment	35
3 Thesis Contributions	39
3.1 P1: Efficiently Guiding Patch Generation Towards Likely Correct Patches	39
3.2 P2: Synthesis and Usage of Differencing Data for Effective Automated Patch Assessment	52
3.3 P3: Making Manual Patch Assessment More Efficient by Facilitating Reasoning about Patch behavior	67
4 Conclusion and Future Work	77
4.1 Summary	77
4.2 Future Work	78
References	81

Part A

Introduction

Software is everywhere, from the vending machine at your local McDonald's to NASA's Curiosity Mars rover, which recently received a major software upgrade [1]. The wide usage of software in various areas has led to the creation of a large industry, in which top companies, like Microsoft, have a market capitalization of trillions of dollars. Given the significant resources dedicated to software development, researchers have been trying for decades to make software programs correct by design [2]. Unfortunately, they have not been successful, new software bugs are still introduced, collected, and studied every day [3]. Therefore, there is no escape from bugs, we have to address them to ensure our programs meet the requirements.

A program that does not comply with given specifications contains a bug, which is an undesirable behavior, security vulnerability, software crash, or failure [4]. To debug a buggy program, it should be modified to comply with the specification. The process of debugging is usually performed manually and consists of multiple iterations of observing hypotheses, making hypotheses, and testing the hypotheses [5]. This heavy involvement of human effort in debugging makes it highly costly [6, 7].

To reduce the costs of debugging, researchers propose automated program repair (APR), a group of techniques that automatically modify a buggy program to meet the given specification [8]. An APR approach automatically finds the faulty part of the code, synthesizes patched versions of the faulty code, validates the patches against the specification, and presents the patches that comply with the specification to the developer. At the end of this process, a developer should review the patched code to select a correct patch. This workflow sounds impeccable, however as of September 2024, there is still no APR tool widely adopted in software industry. The main limitation of existing APR approaches that hinders their usage in industry is their efficiency; after all, they do not significantly reduce the cost of debugging.

There are two main sources of cost, which make APR inefficient. The first

source of APR cost is the *cost of patch generation*. An effective APR approach must generate patches that are highly likely to be *correct*, which means they should implement the intended behavior of the program. This requires extensive static and dynamic analysis of the given buggy program and specification. Such an analysis contains validating the patches against the specification, but can go well beyond it. This analysis is a major source of cost. The second source of APR cost is the *cost of discarding incorrect patches*. As the given specification may not be comprehensive, a patch that is validated against the specification can still be incorrect. Therefore, each generated patch should be further assessed to ensure its correctness. This assessment can be partially automated, however existing APR approaches still require a manual assessment by a code reviewer as well [9]. This means even after using an APR approach, a manual analysis remains essential. Notably, APR approaches often generate a large number of patches that are validated against the specification and manual analysis of such patches can be very costly.

The goal of this thesis is making APR practical by developing novel techniques to overcome APR efficiency obstacles.

1.1 Automated Program Repair

As manual debugging is notoriously costly, APR techniques are suggested to fix bugs automatically. The overall workflow of APR techniques is illustrated in Figure 1.1.

As shown in Figure 1.1, APR consists two major phases: search space exploration and search space analysis. At a high level, we define the search space of APR approaches as follows.

Repair Search Space: Given an APR approach r and a buggy program p , the search space of r is the set of all possible patched versions of p that r can generate.

In the search space exploration phase, the APR approach first takes the buggy program and specification as the inputs and creates a ranked list of suspicious lines (the fault localization component). Then, the APR approach synthesizes patches by modifying the top suspicious lines of the buggy program (the patch synthesis component). The output of this step is the result of the exploration phase: the APR search space consisting of synthesized patches. In the search space analysis phase, the APR approach first validates the synthesized patches against the specification (the patch validation component). The patch synthesis and validation components together form the patch generation step of APR, which outputs a set of validated patches given the buggy program, the specification, and the list of suspicious lines. The validated patches are then automatically assessed to create a ranked list of patches that are marked as possibly correct (the automated patch assessment). Finally, the APR approach enters an interaction

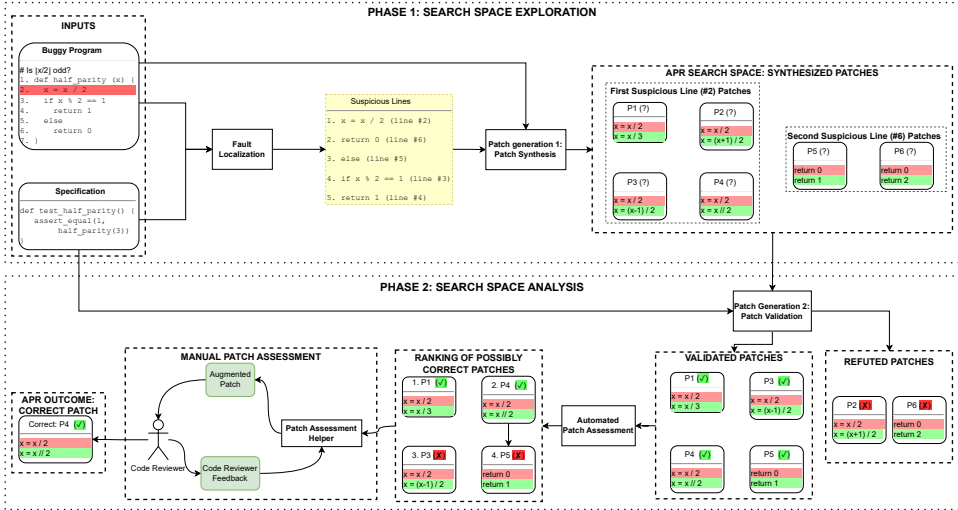


Figure 1.1: An overview of the exploration and analysis of program repair search spaces.

with a code reviewer and presents her with the possibly correct patches until the code reviewer labels a patch as correct (the manual patch assessment component). The patch labeled as correct is the outcome of the APR approach.

To make both phases of APR efficient, we have to reduce the two main types of costs in APR: the cost of patch generation and the cost of discarding incorrect patches. In the following of this section, we quickly review each component of APR with a focus on opportunities to address the main efficiency obstacles.

1.1.1 Fault Localization

The first component of APR is the fault localization component. This component takes the buggy program and specification and outputs a ranked list of suspicious parts of the code. The current state-of-the-art APR approaches employ spectrum-based fault localization (SBFL) technique [10, 11]. This technique particularly suits test-suite based APR, in which the specification is a test case that fails on the buggy program. SBFL assigns a score to each line to show the likelihood of that line causing test failure. There are also APR approaches for warnings reported by program analysis tools, such as SONARJAVA¹ [12]. The reports of program analysis tools identify particular locations of the code that are likely to be buggy. APR approaches that fix such potential bugs directly use the program analysis tool for fault localization.

¹<https://www.sonarsource.com/>

For example, the buggy program in Figure 1.1 is a Python function that takes x and returns 1 if $\lfloor x/2 \rfloor$ is odd. The program is buggy as instead of using the “//” sign for division at line 2 it uses “/”, which means does not compute the floor of the division by 2. The failing test given as part of the specification checks gives $x=3$ as input to the function and asserts that it outputs 1 in response. This test fails as the buggy program wrongly returns 0. Given the buggy program and the specification, the fault localization component creates a ranked list of five suspicious lines, in which line 2 is correctly ranked first. This list is passed to the next components to generate and assess the APR patches.

The spectrum-based fault localization is considered as the standard method in existing APR research. Therefore, in this thesis, we consider fault localization as a fixed component of APR and do not study its efficiency improvement opportunities.

1.1.2 Patch Generation

After the suspicious lines are identified, the APR approach enters its patch generation step. Two components of APR are involved in this step, patch synthesis and patch validation.

The patch synthesis component of APR takes the buggy program and suspicious lines and modifies the suspicious parts of the code. For each suspicious line, a set of novel alternative versions are produced and by replacing the suspicious line with each of the alternative versions a patch is synthesized. For example, in Figure 1.1, four alternatives for the first suspicious line of the buggy program (“ $x=x/2$ ”) are considered: “ $x=x/3$ ”, “ $x=(x+1)/2$ ”, “ $x=(x-1)/2$ ”, and “ $x=x//2$ ”. The first four patches P1-P4 are synthesized using these four alternatives for the second line. Moreover, two alternatives for the second suspicious line (“return 1”) are considered: “return 1” and “return 2”. These two form the next two patches P5-P6. Note that when the time or resources are limited for patch synthesis, the suspicious lines are considered one-by-one according to their ranking until the time or resource limit is exceeded. In our example, only the first two suspicious lines are considered. After the patch synthesis component completes its task the search space exploration phase ends and the result is the search space of the APR approach. In our example, the search space contains P1-P6. At this stage, we do not know if any of these patches are correctly implementing the developers behavior or even if they comply with the given specification. Therefore, we then enter the search space analysis phase in which the correctness of patches is assessed.

The search space analysis phase starts with patch validation, which is the second component involved in the patch generation step of APR. The patch validation component takes the synthesized patches and checks them against the specification. The patches that meet the specification are considered as the validated patches and form the output of the patch generation step of APR. In our Figure 1.1, four patches are successfully validated: P1 (replaces line 2 with “ $x=x/3$ ”), P3 (replaces line 2 with “ $x=(x-1)/2$ ”), P4 (replaces line 2 with “ $x=x//2$ ”), and P5

(replaces line 6 with return 1). All these patches pass the test by returning 1 when 3 is given as input. The result is that the patch generation step of our example outputs four validated patches.

The patch generation step of APR faces an efficiency dilemma. On the one hand, if the search space is large, significant resources should be dedicated to the search space analysis phase to find the correct patch in the large search space. For example, Cardumen [13] has an ultra-large search space, which is likely to contain a correct patch, but requires huge resources to be analyzed. On the other hand, synthesizing a small search space that is likely to contain a correct patch requires intense analysis. For example, constraint-based approaches, such as Nopol [14], aim at synthesizing a search space that only contains patches that are guaranteed to meet the specification. For this purpose, they employ the notoriously resource demanding constraint solvers. *In sum, an efficient patch generation technique should take minimal resources to synthesize a search space with highly likely patches.*

1.1.3 Automated Patch Assessment

The automated patch assessment component takes the next step of the search space analysis phase. This component is mainly responsible to automatically mitigate the overfitting problem. To this end, it recognizes the differences between the patches and guides the manual patch assessment based on these differences. The goal is making it possible to find a correct patch by manually analyzing as few validated patches as possible.

Automated patch assessment is performed using the data gained through static or dynamic analysis of patches. Based on this data, three types of actions are taken to automatically differentiate between the patches and minimize the number of patches to be manually analyzed. 1) A group of patches are classified as overfitting [15]. The patches labeled as overfitting are not manually analyzed. 2) The remaining patches are clustered into groups of similar patches [9]. As the patches in each group are considered to be similar, only one patch from each group has to be manually analyzed. 3) The patches are ranked according to their likelihood of being correct [16]. The patches are manually analyzed one-by-one ordered by their rank; once a patch is labeled as correct the remaining patches are not manually analyzed. The automated patch assessment component of an APR approach takes one or multiple of these three actions to cut down the required manual analysis. The output of this component is a ranked list of validated patches that are labeled as possibly correct.

In our Figure 1.1 example, the automated patch assessment component classifies two of the validated patches as incorrect: P3 (replacing line 2 with “ $x=(x-1)/2$ ”) and P5 (replacing line 6 with “return 1”). The other two validated patches are classified as possibly correct. The top ranked patch replaces line 2 with “ $x=x/3$ ” and the second one replaces “ $x=x/2$ ”. This means only these two

patches require manual analysis and if the first one is correct, the second one is not manually analyzed anymore.

The automated patch assessment component of APR plays a crucial role in making APR efficient. The more accurate and effective the automated patch assessment is, the less time and resources are required for the manual patch assessment. Also, note that the automated patch assessment may require large resources itself. Therefore, an efficient APR approach obtains data useful for patch differentiation with minimal cost and employs it with maximal effectiveness.

1.1.4 Manual Patch Assessment

The manual patch assessment component takes the final step of APR. In this component, the APR approach enters an interaction with a code reviewer, who goes through the ranking of patches to find a correct patch. This interaction happens iteratively between a patch assessment helper unit and the code reviewer. At each iteration, the patch assessment helper augments the patch with extra information useful for review and presents it to the code reviewer in an effective manner. The code reviewer should understand the behavior of the patch and give feedback whether it corresponds with the intended behavior. For example, researchers have studied the impact of adding natural language explanation of the patch on the code review process [17]. As another example, the effectiveness of integrating the manual patch assessment into common development environments, such as IDEs, is investigated by APR researchers [18].

In the Figure 1.1 example, the code reviewer has to first analyze P1 (“ $x=x/3$ ”) and then P4 (“ $x=x/2$ ”), as P1 is not correct. After the code reviewer analyzes P4 and confirms its behavior complies with expectations, the APR process ends successfully and P4 is reported as the APR outcome.

An efficient APR approach should make it as smooth as possible for code reviewers to understand the behavior of the patch. This requires presenting useful and easy to understand information about the behavior of the patch that goes beyond its plain text.

1.2 Problem Statement

Per our overview of the APR workflow, we identify three main problems that make APR inefficient. These three problems correspond to three APR components as follows.

1.2.1 P1: Inefficiencies of APR Patch Generation

P1: APR approaches spend large computation resources for generating patches that are unlikely to be correct.

The problem of inefficiency in APR starts from the patch generation step. The patch synthesis spends significant resources to produce the APR search space.

This search space may contain many patches that do not even meet the specification, let alone correctly implementing the intended behavior [13]. This leads to wasting the resources dedicated to synthesize and assess these patches. To guarantee that such patches are not synthesized, extra analysis of the buggy program and specification is required. Such an analysis can be highly costly, as seen in constrained-based APR approaches, like Nopol [14]. An efficient patch generation technique spends little resources for patch synthesis and generates mostly correct patches.

1.2.2 P2: Inefficiencies of APR Automated Patch Assessment

P2: Effective automated patch assessment depends on the availability of elusive information that distinguishes between patches.

The task of the automated patch assessment component can be seen as distinguishing between various program versions, including validated patches and the buggy program. First, a group of the patches should be separated from the others and discarded as overfitting patches. Secondly, different clusters of patches should be detected and just one representative from each cluster should be manually assessed. Finally, the patches that require manual assessment should be assigned different rankings according to their correctness likelihood. All these steps depend on obtaining and effectively using data that marks the differences between the validated patches. Such data can be either collected from existing software corpus or generated for specific given patches. For example, open-source repositories contain millions of successful and failed bug-fixing commits. These commits provide valuable information about the differences between patches that successfully fix a bug and the ones that fail to do so. APR efficiency can be improved by employing such historical data. Researchers have also used traditional techniques, such as test generation with EvoSuite [19], to produce patch differencing data [15]. However, these techniques are not particularly designed or optimized for differentiating between APR patches. To make APR efficient, we can create novel techniques that are fine-tuned to generate effective patch differencing data.

1.2.3 P3: Inefficiencies of APR Manual Patch Assessment

P3: In manual patch assessment, reasoning about patch behavior is notoriously time-consuming.

Inefficiencies in manual patch assessment are particularly important, as it is the only APR component requiring human expertise. The human expertise is needed to understand the behavioral differences between the buggy and patched programs. The behavioral differences are not clearly given in the plain text diff between two program versions. This means an efficient APR approach should augment the textual code diff with extra information that clarifies the behavioral differences. Note that the behavioral differences are widespread and appear at

different locations with various levels of granularity. To make the patch easy to understand, the behavioral differences should be presented concisely and in a human-readable format. Even though code review assistance is a well researched area [20], helping the review of APR patches is not widely investigated yet.

1.2.4 Summary of Problem Statements

To sum up, we consider the three following problems with the efficiency of APR search space exploration and analysis.

- P1: APR approaches spend large computation resources for generating patches that are unlikely to be correct. An efficient APR approach guides the patch generation process towards likely correct patches, while using minimal resources.
- P2: Effective automated patch assessment depends on the availability of elusive information that distinguishes between patches. An efficient APR approach needs methods for finding differences between generated patches and prioritizing them.
- P3: In manual patch assessment, reasoning about patch behavior is notoriously time-consuming. An efficient APR approach needs to present the behavioral information of the patch in a suitable and concise manner.

1.3 Thesis Contributions

In this thesis, we discuss our contributions in response to each of the APR efficiency problems.

1.3.1 Answer to P1: Making Patch Generation Efficient

We make two contributions aimed at improving patch generation efficiency. Both of these contributions focus on generating patches that are likely to be correct with minimal cost. The first contribution proposes SORALD, a template-based APR tool for fixing SONARJAVA static warnings. SORALD uses official documentation written by domain experts to design highly accurate fix templates. The second contribution introduces CIGAR, an LLM-based APR tool that fixes test failure causing bugs. CIGAR employs three delicately designed prompts to guide the LLM toward generating a diverse set of patches that are validated against the specification. CIGAR also uses prompting strategies to minimize the cost of using LLMs. We now delve into the details of these contributions.

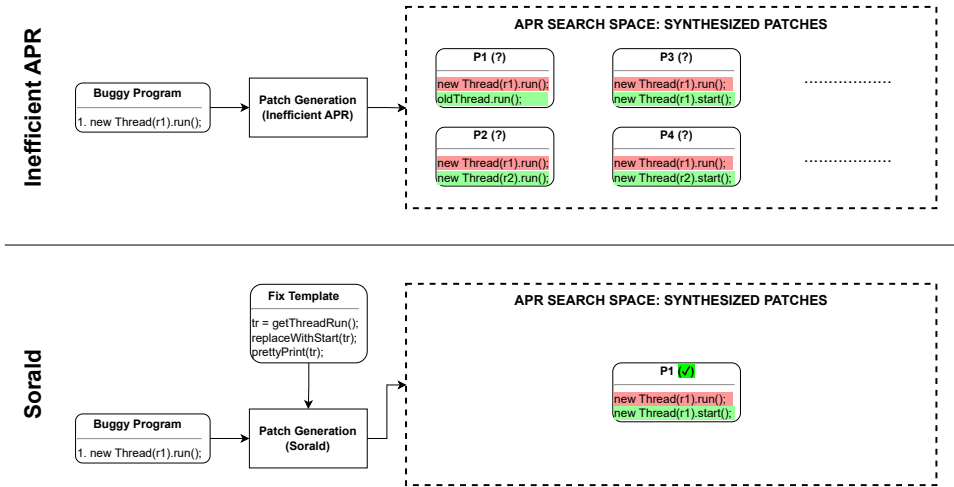


Figure 1.2: SORALD as an efficient template-based APR tool for static warnings.

Contribution 1: A Highly Precise Template-based Approach for Fixing Static Analyzer Issues

The first contribution of this thesis [12] is SORALD. SORALD is a template-based APR tool for fixing SONARJAVA static warnings. SONARJAVA is a static analyzer for Java that checks a given program against a set of code quality rules. If the program violates a rule, SONARJAVA reports an issue. To fix these issues, SORALD uses its fix templates that apply pre-defined modifications on the abstract syntax tree (AST) of the buggy program. These modifications make the program in compliance with the violated rule, while also implementing the intended behavior. SORALD makes APR patch generation efficient by employing highly accurate fix templates. These templates are designed according to the official documentation provided by SONARJAVA. For any issue reported by SONARJAVA, SORALD applies the template corresponding to the violated rule and generates exactly one patch. As SORALD fix templates carefully follow the official suggestions by SONARJAVA experts, they tend to generate likely correct patches. Moreover, as SORALD templates generate a search space with just one patch, the search space analysis phase is straightforward and does not require too much effort.

Figure 1.2 illustrates why SORALD is an efficient APR tool. The rule S1217 in SONARJAVA indicates that “Thread.run()” should not be called, instead “Thread.start()” should be used. An inefficient APR tool takes a buggy statement “new Thread(r1).run()” and tries various modifications to synthesize its search space. The result is a large search space with many patches that there is not reason for their correctness. In contrast, SORALD uses its accurate fix template taken inspired by the official documentation and generates only one patch. This patch exactly applies the suggested fix and replaces “run()” with “start()”. Consequently, SORALD generates its

search space fast and produces only one patch that is highly likely to be correct according to the domain experts. Note that in the next steps of SORALD APR, only one patch has to be analyzed, which contributes to SORALD efficiency.

We evaluate SORALD on a dataset of 161 Java projects on GitHub. We use SORALD to fix violations of 10 SONARJAVA rules. SORALD tries to fix 1,307 rule violations and successfully generates a correct patch for 65% (852/1,307) of them, which indicates a high effectiveness compared to APR literature standards [21]. To fix each of This shows that SORALD is an effective APR tool, besides being efficient due to its small and accurate search space.

Contribution 2: An LLM-based APR Approach with a Highly Efficient Search Space Exploration Strategy

In the second contribution of this thesis, we introduce CIGAR. CIGAR iteratively utilizes LLMs to fix test-suite based bugs in three steps. First, it sends the LLM an *initiation prompt* that contains the bug and its test failure. CIGAR requests the LLM to generate a bug-fixing patch. If the LLM fails in generating a patch that passes the test, CIGAR prompts the LLM with an *improvement prompt*, containing the bug, a summary of previously generated patches and their corresponding test failures. The patch generation continues with new iterations of improvement until a plausible patch is generated. CIGAR also employs a reboot strategy to ensure the efficiency of its patch generation. According to this strategy, CIGAR reboots the improvement process after every few iterations and starts the repair process from the scratch. The reason is that per our experiments continuing improvement for too many iterations makes the LLM generate repetitive patches that are of no use. Finally, after the generation of a plausible patch, CIGAR sends *multiplication prompts* to the LLM to generate alternative plausible patches. The multiplication prompts contain a summary of previously generated plausible patches to avoid spending resources for generating repetitive patches.

CIGAR’s three types of prompts and its reboot strategy are delicately designed to build and improve the search space step-by-step. The feedback on previous patches that is given to the LLM at each step guides the model toward generating new patches that are more likely to be correct. Figure 1.3 depicts how CIGAR’s patch generation is more efficient compared to traditional APR approaches. An inefficient APR tool takes the inputs and produces a huge search space of patches in one go. Many of the patches its search space do not comply with the specification and have to be later discarded. In contrast, CIGAR works iteratively and step-by-step. In its first iteration, it generates just a few patches. This means CIGAR does not spend resources to generate patches at the beginning, when very limited information is available. Next, CIGAR takes the initially generated patches and generates new ones that are different from previous ones (iteration 2). At every iteration, new information regarding previous patches is added to the prompt to better guide the LLM. Once a plausible patch is generated (iteration

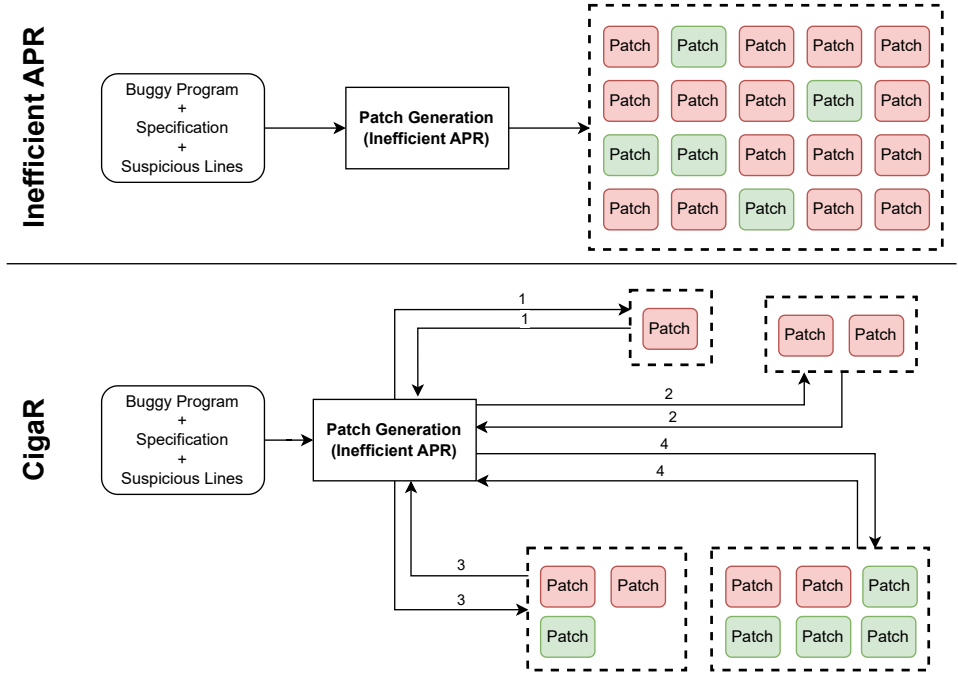


Figure 1.3: SORALD as an efficient template-based APR tool for static warnings.

3), CIGAR requests the LLM to multiply it by generating new distinct plausible patches (iteration 4). The result of this process is a smaller search space, compared to an inefficient APR tool, which means CIGAR spends less resources for patch generation. Moreover, CIGAR’s search space has a high ratio of plausible and correct patches, which makes the search space analysis phase more efficient.

We evaluate CIGAR on two widely used datasets of Java bugs: 162 bugs from HUMAN-EVAL-JAVA and 267 bugs from DEFECTS4J. We measure the efficiency of CIGAR against its state-of-the-art competitor CHATREPAIR in terms of the total number of tokens sent to and received from the LLM. On average, CIGAR spends 127k tokens per bug while the CHATREPAIR uses 467k tokens per bug, indicating a 73% cost reduction by CIGAR. We also show that while significantly reducing the costs, CIGAR outperforms the state-of-the-art APR tools by fixing 39.8% (171/429) of all considered bugs.

1.3.2 P2: Synthesis and Usage of Differencing Data for Automated Patch Assessment

We make two contributions in answer to address P2. The first contribution is LIGHTER, a tool that employs historical open-source data to estimate the po-

tential of fix templates used by template-based APR approaches. The second contribution is MOKAV, an LLM-based tool to generate test inputs that differentiates between the functionality of patches.

Contribution 3: Estimating the Potential of Program Repair Fix Templates with Commit Analysis

In our third contribution, we propose LIGHTER, a lightweight tool that employs historical bug-fixing data to rank template-based APR patches. A template-based APR tool applies various pre-defined sets of modifications, called fix templates, to repair buggy programs. LIGHTER provides a framework to readily encode the search space produced by fix templates and compare them against human-made commits. The result of this comparison shows how much each fix template follows the modification patterns used by developers. Given the human expertise encoded in open-source commits, a fix template that generates patches similar to human-made commits has a strong potential to generate valuable patches. Therefore, LIGHTER can be used to rank patches of template-based APR tools according to the potential of the template used for their generation.

LIGHTER works as follows. First, it uses the *change pattern specification language* to specify the modifications pattern of a fix template *ft*. Next, it goes over human-made commits and uses Spoon [22] to extract the AST modifications for each commit. Finally, LIGHTER checks whether AST modifications by a commit *c* follow the modifications pattern of *ft*. If it does, we say that *ft* covers *c*. A fix template that covers many commits is likely to generate useful patches.

Figure 1.4 illustrates how LIGHTER makes the automated patch assessment more efficient at a high level. The figure shows an example where the search space contains 20 patches among which five are correct. These patches are generated by four different fix templates. The ranking of patches given by an inefficient automated patch assessment puts the correct patches at ranks 8, 9, 11, 13, and 14. In contrast, LIGHTER uses historical human-made commits to find the more effective fix templates. Next, it ranks the patches according to the template that generates them: patches generated by “Template 1” are ranked 1st to 6th, patches by “Template 2” are ranked 7th to 10th, patches generated by “Template 3” are ranked 11th to 14th, and patches by “Template 4” are ranked 15th to 20th. The result is that patches 2, 3, 4, 9, and 10 are correct. Consequently, when LIGHTER is used, the code reviewer finds the correct patch with four manual assessments, compared to eight manual assessments when inefficient APR approach is used.

We evaluate LIGHTER on a dataset of 7,583 commits from 72 open-source Java projects. In this experiment, we use LIGHTER to specify the fix templates of eight well-known template-based APR tools: Arja [23], Cardumen [13], Elixir [24], GenProg [25], jMutRepair [26], Kali [27], Nopol [14], and NPEfix [28]. We 9.85% (747/7,583) of the considered human-made commits follow the modifications pattern of a least one of the considered fix templates. Overall, Elixir has most human-made commits similar to its fix template patches, by covering 4.86%

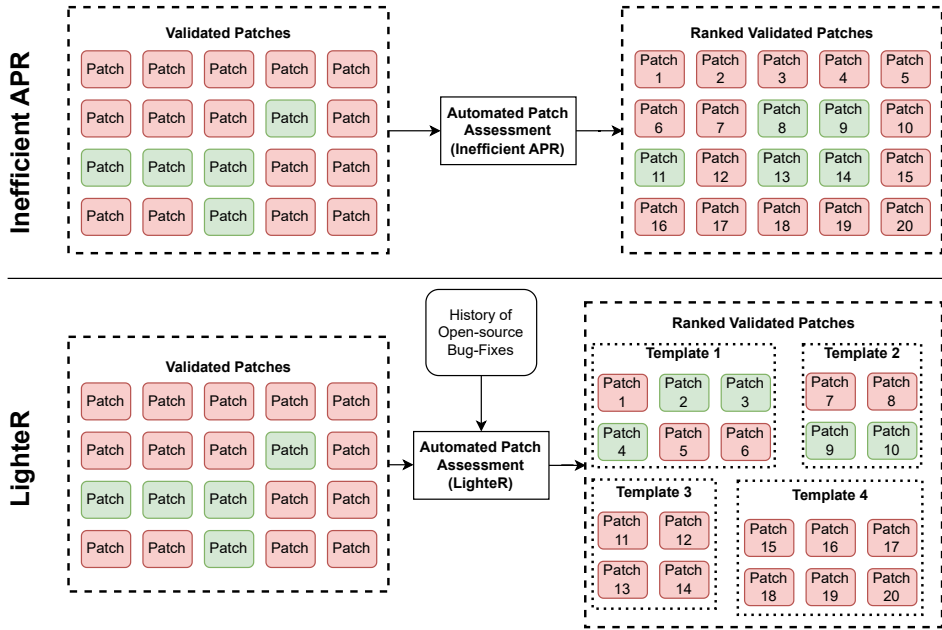


Figure 1.4: LIGHTER: a lightweight tool for estimating the potential of APR fix templates.

(369/7,583) of the commits. The experiment shows that LIGHTER uses historical data to estimate the potential of fix templates. This contributes to efficient automated patch assessment.

Contribution 4: Execution-driven Differential Testing with LLMs

The fourth contribution of this thesis introduces MOKAV. MOKAV is an LLM-based tool that generates functional difference exposing tests (DETs), test inputs that produce different outputs on different versions of a program. These tests are useful for clustering APR patches. In patch clustering studies, the patches that produce similar outputs on generated tests are clustered into the same group [9]. As patches in the same cluster are considered to be functionally similar, only one representative from each cluster needs to be manually assessed. This makes APR significantly more efficient by reducing the number of patches that need manual assessment.

MOKAV works iteratively. Given two versions of a program P & Q, MOKAV runs an existing test input, that is not difference exposing, on them and collects their execution traces. Then, it sends the P & Q together with collected execution traces to an LLM and asks it to generate tests that expose functional differences. If the LLM fails to generate DETs, MOKAV iteratively gives feedback to the LLM

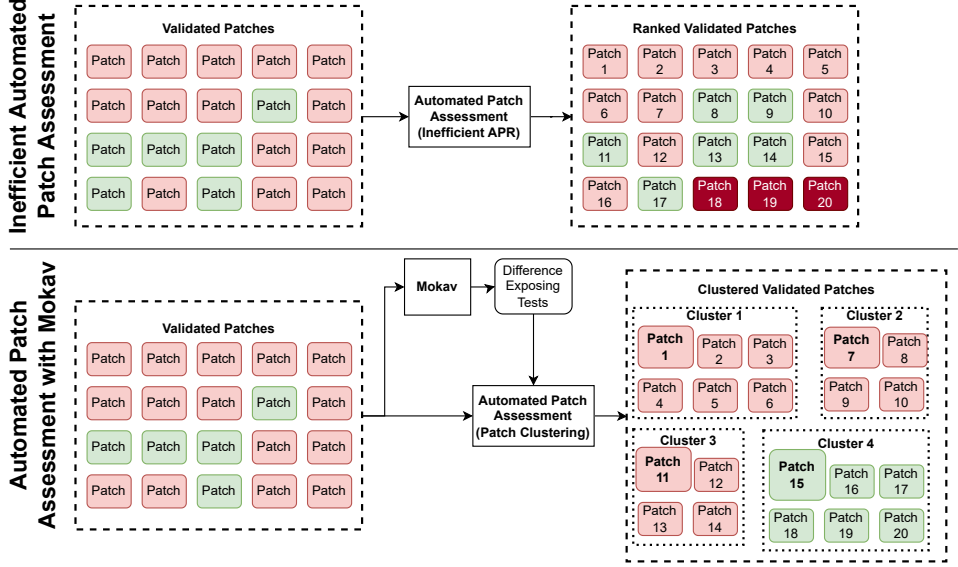


Figure 1.5: MOKAV: an LLM-based tool for generating difference exposing tests.

until a DET is generated.

Figure 1.5 shows a high level example of how MOKAV makes APR more efficient. In this example, 20 validated patches are given among which six are correct. The inefficient automated patch assessment technique labels three patches as overfitting and ranks the remaining 17. The correct patches are ranked 8, 9, 11, 13, 14, and 17. This means the code reviewer needs to manually assess the first eight patches until she finds a correct patch. In contrast, the automated patch assessment with MOKAV first employs MOKAV to generate DETs. Then, these DETs are given to the automated patch assessment which clusters the 20 validated patches into four clusters. The patches in clusters 1-3 are overfitting and the patches in cluster 4 are correct. The code reviewer manually assesses only one representative patch from each cluster, which leads to the manual assessment of four patches: patches 1, 7, 11, and 15. Therefore, the code reviewer is able to find a correct patch by assessing only four patches. This example shows how MOKAV helps to reduce the number of manually assessed patches from eight to four.

We evaluate MOKAV on two datasets of bug-fixing code changes: QuixBugs [29] with 32 pairs of <buggy, fixed> Python programs and C4DET, which is our curated dataset of 1,535 pairs of <buggy, fixed> Python programs. In this experiment, MOKAV generates DETs for 100% (32/32) of QuixBugs pairs and 81.7% (1,255/1,535) of C4DET pairs. This shows MOKAV outperforms the baselines, PYNGUIN and DPP, as they generate DETs for 50% (16/32) of QuixBugs pairs

and 37.3% (573/1,535) of C4DET pairs. We also notice that MOKAV’s execution-driven iterative approach improves its effectiveness significantly. Overall, our evaluation show that MOKAV is an effective tool for generating DETs. These DETs can be used for making APR more efficient with patch clustering in the future.

1.3.3 Answer to P3: Helping Code Reviewers to Better Understand Behavioral Differences

To make manual patch assessment more efficient, we introduce COLLECTOR-SAHAB. COLLECTOR-SAHAB augments code diffs with runtime information to make it easier for code reviewers to understand the behavioral differences made by a patch.

Contribution 5: Augmenting Patches with Human-readable Runtime Information

In our fifth contribution, we design and implement COLLECTOR-SAHAB, a tool for augmenting code diffs with human-readable runtime information. COLLECTOR-SAHAB takes two versions of a program P & Q and a test on which P & Q behave differently. The outcome of COLLECTOR-SAHAB is the code diff between P & Q augmented with a concise runtime difference information. To compute the runtime difference, COLLECTOR-SAHAB runs the text on P & Q and collects the variable values and return values during the test execution. Next, it detects the differences between the collected information for P & Q. Finally, a small subset of the detected differences is selected and added to the code diff. COLLECTOR-SAHAB’s augmented code diff is integrated into the widely-used GitHub UI to make the patch assessment environment more familiar and comfortable for code reviewers.

Figure 1.6 shows how COLLECTOR-SAHAB makes manual patch assessment more efficient. An inefficient manual patch assessment technique simply computes the code diffs for the patches one-by-one and presents them to the code reviewer. In contrast, a manual patch assessment component with COLLECTOR-SAHAB first takes the patch, the buggy version, and a test and identifies runtime differences. Next, a code diff augments generates the augmented diffs and presents them to the code reviewer one-by-one. The augmented diff contains concise information regarding behavioral differences between the patch and the buggy version. This makes understanding the behavioral changes easier for code reviewers.

We evaluate COLLECTOR-SAHAB on a dataset of 584 plausible APR patches for Defects4J [30] bugs. Note that a plausible patch passes the test that the buggy version fails on. This guarantees the plausible patch causes behavioral differences. The experiment shows that COLLECTOR-SAHAB detects a runtime difference for 94.8% (554/584) of plausible patches, which means COLLECTOR-SAHAB very rarely misses behavioral changes. We also conduct a user study and

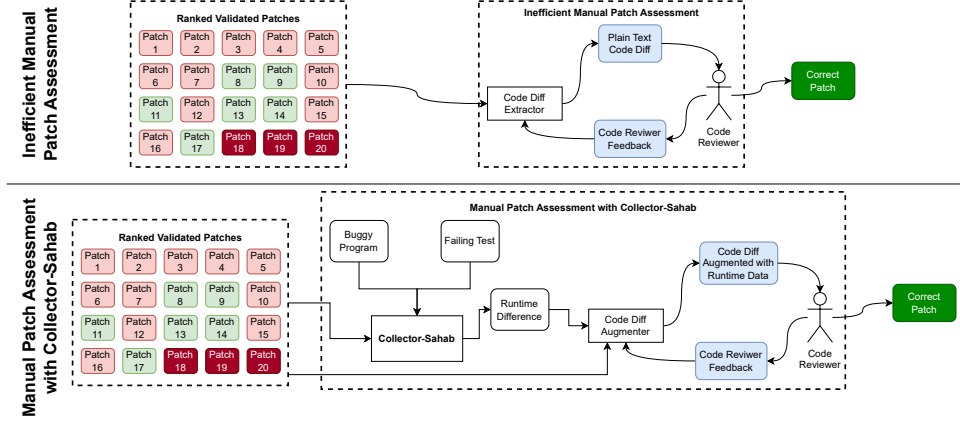


Figure 1.6: COLLECTOR-SAHAB: a tool for augmenting patches with runtime difference information.

show that developers find COLLECTOR-SAHAB code diff augmentations useful, clear, and novel. Overall, COLLECTOR-SAHAB improves manual patch assessment efficiency by making it easier for code reviewers to understand behavioral changes.

1.4 Thesis Outline

In chapter 2 of this paper, we review the state-of-the-art research on automated program repair from efficiency aspect. In chapter 3, we discuss the details of our contributions on the topic of APR efficiency. Finally, chapter 4 concludes this thesis and presents potential future work. The rest of this compilation thesis contains the papers that present our contributions.

State of the Art

In this chapter, we review the literature on different steps of automated program repair [8, 31, 32] with a focus on the efficiency of existing approaches.

2.1 Automated Program Repair Workflow

Figure 2.1 overviews the workflow of APR. The input to an APR approach is a buggy program and a specification that specifies the expected behavior and qualities of the program. In this thesis, we consider two common types of specification: test suites and static analyzer requirements. The program’s failure to meet the specification is considered to be the bug. An APR approach outputs a correct patch that fixes the bug. We split the process of APR into four steps as follows.

1. Fault Localization: The very first step of APR is finding the faulty parts of the code that. For this, at first the buggy program is checked against the specification. For example, when the specification is a test suite, it is executed on the buggy program. This execution verifies the main assumption of APR applicability: the presence of a failing test that exposes the bug [33]. To facilitate scientific investigations, researchers have created datasets of real-world buggy programs that guarantee the presence of such tests [3, 30, 34, 35, 36, 37, 38]. This check against the specification makes the APR process more efficient by early elimination of unamenable bugs.

After the existence of a fault is verified, the APR approach should identify the code locations that potentially cause the bug [33]. This task is called fault localization. While various fault localization techniques are introduced for APR [39, 40, 41], the state of the art method for test suite based bugs is spectrum-based fault localization (SFL). In SFL the program statements are ranked based on their suspiciousness [42]. The suspiciousness of a statement is estimated by its dynamic association with failing tests [43]. The result of fault localization is

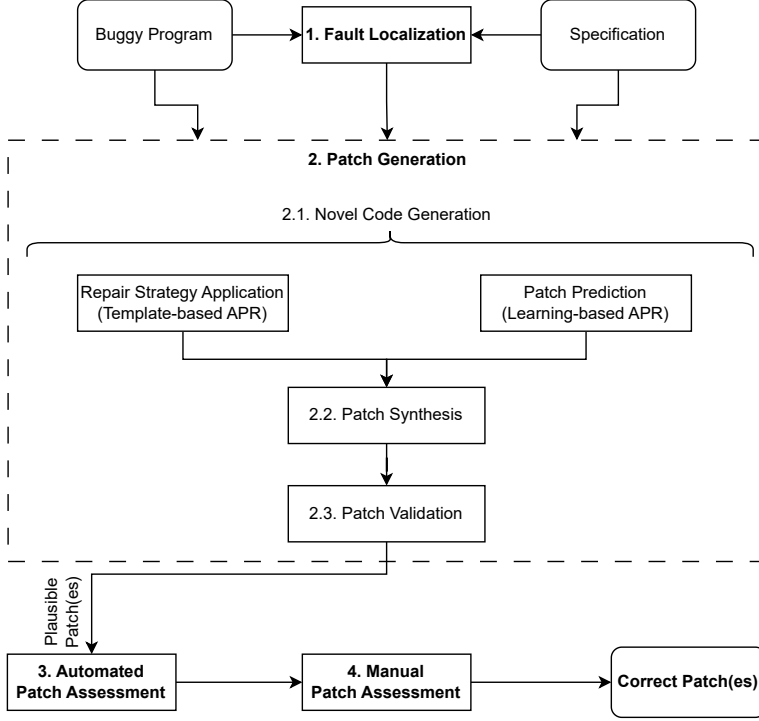


Figure 2.1: An overview of the test-suite automated program repair process.

the ranking of suspicious statements. This improves the efficiency of the APR approach by concentrating all the resources on the few lines that are likely to contain the bug. For bugs detected by static analyzers, the location of the bug is usually marked by the static analyzer [12].

2. Patch Generation: In the third step of APR, called patch generation, the APR approach removes, adds or replaces statements in suspicious locations. The patch generation procedure splits into three main phases. First, novel code components are made to replace the buggy code (novel code generation). We discuss the state of the art methods for making novel code components in details. In the second phase of patch generation, the buggy code and the new code components are merged together to synthesize patched programs (patch synthesis). As multiple novel code components can be produced in the first phase, multiple patches can be synthesized in the second phase as well. Finally, in the third phase of patch generation, all synthesized patches are validated against the given specification (patch validation). The successfully validated patches are called *plausible patches* [32] and comprise the output of the patch generation step. The patch generation step has a high computation [44, 45] cost and produces the whole set of patches

that should be later assessed either automatically or manually. This makes patch generation a core target of APR efficiency improvement techniques.

3. Automated Patch Assessment: While the plausible patches produced in the previous step pass all the tests, they are not guaranteed to correctly implement the expected behavior. Therefore, the APR process includes an automated patch assessment to identify the *correct patches*: patches that fix the bug by correctly by implementing the expected behavior. The automated process of identifying correct patches among plausible ones is called *overfitting detection* [15]. Overfitting detection plays an integral role in making APR efficient by removing incorrect patches that have to be manually analyzed otherwise.

4. Manual Patch Assessment: In the final step, the plausible patches that have successfully passed the automated patch assessment step, are manually analyzed to ensure they are correct. Manual analysis of patches is notoriously complex and time-consuming [18]. To alleviate this difficulty of this task, various techniques and tools are proposed to help developers through the patch reviewing process.

Based our description of the APR workflow, we define the repair search space [46] as follows.

Repair Search Space: Given an APR approach r and a buggy program p , the repair search space of r on p is the set of all patches r produces for p through its fault localization and patch generation phases.

The goal of this thesis is to make APR efficient with a focus on how the repair search space is shaped, explored and assessed. Before discussing the our contributions in chapter 3, in the remaining of this chapter, we review the existing work on patch generation, automated patch assessment, and manual patch assessment. We do no further discuss fault localization, as most APR approaches use standard methods for this step.

2.2 Patch Generation Techniques

As explained in section 2.1, patch generation starts with producing novel code components that should replace the old code. As patch synthesis and patch validation are deterministic, novel code component generation is the main phase that sets the type and efficiency of APR patch generation. Based on this observation, there are three types of patch generation techniques: template-based patch generation, constraint-based patch generation, and learning-based patch generation [47]. We now review these three types of techniques as follows.

2.2.1 Template-based Patch Generation

The early APR approaches, such as GenProg [25, 48], use template-based techniques for patch generation. A template-based APR approach searches for a correct patch among a predefined set of patches [47]. The predefined set of patches is determined by the repair strategies that the APR approach applies on the buggy program. For example, removing a statement by replacing it with an empty code component is a strategy employed by various APR tools [49], such as Kali [27] and Arja [23]. A template-based APR applies its repair strategies one-by-one to generate the patches. At each step, the generated patch is validated against the specification and the repair process stops when a patch is successfully validated. Based on this repair procedure, template-based patch generation is also called as generation and validate (G&V).

Heuristic-based APR: The first template-based approaches employed heuristic strategies for repair. In 2009, Weimer et al. [50] proposed using genetic programming (GP) [51] to generate bug-fixing patches for the first time. This proposal was later evolved into GenProg [25, 48, 52, 53]. GenProg generates new variants of a buggy program by adding, removing, or replacing statements that are executed by failing tests. The variants are then gradually evolved through mutation and crossover operations until one of the variants passes all the tests. In its GP algorithm, GenProg uses a simple fitness function based on the number of passed and failed tests. To keep the size of the search space feasible for exploration, GenProg only uses existing statements in the code for adding or replacing faulty code. This is based on the hypothesis that “a program that contains an error in one area likely implements the correct behavior elsewhere [54]”. This design decision by GenProg shows the efficiency challenge for template-based APR: considering too many different novel code components is infeasible for exploration, while limiting the search space is a threat to the repair power.

Based on the ideas proposed by GenProg, many other APR approaches are introduced that use genetic programming and other heuristic strategies [55, 56, 57, 58]. Fast et al. [59] incorporate test selection and formal specification into the fitness function to improve the efficiency and precision of search space exploration. Qi et al. [60] propose TrpAutoRepair. TrpAutoRepair employs fault-recorded test case prioritization to rank the tests that the patches are validated against. With this test prioritization, TrpAutoRepair significantly improves the efficiency of patch validation. Schulte et al. [61, 62] apply an approach similar to GenProg on binary level defects. To improve the efficiency, they use a stochastic sampling for localization.

Researchers have also tried to improve the original GenProg technique by using different search strategies. Qi et al. [63] question the effectiveness of genetic algorithm employed by GenProg and use a random selection instead. Interestingly, they show this simple method, implemented in RSRepair, outperforms GenProg in terms of the number of required test executions and patch generation

```

1 Set<E> removed = getAllEdges(sourceVertex, targetVertex);
2 if (removed == null) {
3     return null;
4 }
5 removeAllEdges(removed);

```

Listing 2.1: A correct patch generated by NPEfix for the buggy code from *JgraphT* project at commit “fd759d76”.

trials. Yuan et al. propose Arja [23], a GP-based APR approach for Java that optimizes two objectives at the same time: the rate of failing tests and the size of the patch. Their experiments show that Arja significantly outperforms jGenProg [64], the implementation of GenProg for Java applications, on the Defects4J [30] dataset. Wen et al. introduce CapGen [65], another APR tool for Java bugs. CapGen modifies the code at a more fine-grained level, compared to GenProg, by considering types and expressions in addition to statements. It also utilizes a context-aware prioritization of mutation operators and patches to make the search space exploration more efficient.

One of the major steps in GenProg-like APR approaches is to select a proper *fix ingredient*. A fix ingredient is a code component in the existing program that is used to replace the buggy code. Considering all program statements for ingredient selection makes the search space too large. To tackle this issue and improve the search efficiency, Ji et al. [66] implement SCRepair. SCRepair uses reusability metrics for similarity to prioritize the fix ingredients that are similar to but not the same as the buggy code. Wang et al. [67] also use similarity metrics to choose the fix ingredients, but they search for the ingredient in open-source repositories. ssFix [68] also uses similarity metrics to search in a given code base and find appropriate fix ingredients similar to the faulty code and its surrounding context. Xin et al. [69] later improve ssFix by using statement-level code blocks for local search and method-level code for global search. Jiang et al. introduce SimFix [70], which creates an abstract search space from the existing patches and then concretizes it using similar code.

APR with Manually Crafted Templates: There is a large number of template-based APR approaches that employ predefined fix templates to modify the buggy program. These approaches are also called pattern-based APR methods [47, 71, 72, 73]. The strength of this technique is using high-quality modification patterns that are likely to fix the buggy program. This makes the APR approach efficient by limiting the search space to patches that are carefully design for fixing the given bug.

NPEfix [28] is one of the early such template-based APR approaches, specifically designed to fix `NullPointerExceptions`. NPEfix employs five types of fix

templates as its repair strategies to handle `NullPointerExceptions`. The fix templates include adding null checkers for suspicious statements and skipping those statements or replacing their expressions with new ones. For example, Listing 2.1 shows a patch generated by NPEfix [28] for a buggy code from the Jgrapht¹ project. The original buggy code retrieves all the edges from source to target (line 1) and then removes the retrieved edges (line 5). The program faces a `NullPointerException` when the set of retrieved edges in variable `removed` is null. To fix this issue, the NPEfix APR approach [28] applies a fix template that adds a null checker before using a variable. In this case, the null checker inspects the value of `removed` and if it is null, returns null and does not execute the rest of the method. This template is carefully designed to fix `NullPointerExceptions` and limits the search space to a very limited set of possible patches.

Kim et al. introduce PAR [74]. They manually study 60,000 open-source patches and find 10 fix template accordingly. These templates guide APR towards producing patches that are similar to human-made fixes. Tan and Roychoudhury [75] follow a similar method for template design. They analyze 73 open-source regressions and come up with eight fix patterns for repairing regression bugs.

Qi et al. [27] create Kali as a simple APR approach that just removes functionality using three templates: removing a statement, changing an if condition to true or false, and returning from a method early. The authors interestingly find that this simple approach outperforms GenProg in terms of the number of correct fixes. jMutRepair [26] is another APR approach that uses a simple fix template by changing a unary or binary operator inside an if condition. This simple template is also able to generate plausible patches for 29 Defects4J bugs.

Martinez and Monperrus propose Cardumen [13]. Cardumen replaces existing expressions in suspicious lines with new expressions. The new expressions are generated in two steps: an existing expression is selected and then the variables and literals are replaced with other variables and literals from the code. This approach makes the patch generation highly flexible, however it leads to an ultra-large search space. Cardumen reduces the search space size by filtering out the new expressions with a wrong type. To navigate the search space more efficiently, Cardumen builds a probabilistic model of variable names and prioritize expressions that use variables likely to occur together. Cardumen successfully fixes eight Defects4J bugs that were not previously fixed by any APR approach.

Saha et al. [24] introduce Elixir, an APR approach that employs multiple fix templates including templates particularly designed for object oriented programming languages. Elixir replaces existing expressions with new ones, where the new expressions can contain method calls on objects in the scope. The large list of possible patches is ranked using a machine-learning model. This model considers the appearance frequency of identifiers in the new expression, the location where those identifiers are taken from, the relevance of the identifiers to the bug report, and the similarity of the identifier to the tokens in the context.

¹<https://github.com/jgrapht/jgrapht>

Elixir shows state of the art performance with its flexible search space creation and efficient search space exploration.

Mining templates: Researchers have also tried various methods for mining useful fix templates from the large corpus of available software projects. Le et al. [76] create HDRepair. This APR approach works in three phases. First, it investigates the history of considered projects to find fix templates using graph mining techniques. Then, it applies existing mutation operators to generate patches. Finally, it prioritizes patches that follow more frequent fix templates. Long et al. [77] introduce Genesis, an APR approach that extracts templates from existing patches. By analyzing 577 null pointer, class cast, and out of bound historical bug-fixing patches, Genesis infers 108 templates. Using these templates, Genesis fixes 21 of 49 defects collected from open-source projects. To make the search space exploration efficient, Genesis formulates the exploration as an integer linear program (ILP) problem. This ILP problem maximizes the number of covered historical patches and bounds the number of patches in the search space.

Researchers have used machine learning methods, such as clustering, to extract change templates [78]. Koyuncu et al. [79] use iterative clustering on atomic changes to extract fix templates from a given set of patches. They apply their method on 11,416 patches and incorporate their extracted templates in an APR approach, called PARFixMiner. PARFixMiner fixes 26 bugs from Defects4J and 81% of its plausible patches are also correct. Liu et al. [80] propose AVATAR. They first use convolutional neural networks to extract features from violations of FindBugs [81] static analyzer and fixes for those violations. Based on the features, then they cluster the fixes and extract fix templates from them. The extracted fix templates are implemented in the AVATAR APR approach. Interestingly, they find that the templates extracted from fixes for FindBugs static warnings are applicable on Defects4J test failure bugs, AVATAR fixes 34 of Defects4J bugs.

Liu et al. [82] conduct an extensive study of existing template-based APR approaches. By considering 17 approaches from the literature, they identify 35 fix templates and split them into 15 categories. They implement these templates in their tool, called TBar. TBar is able to fix 43 of Defects4J bugs, which is the best performance a template-based has achieved.

Constrained-based Patch Generation: A major drawback of aforementioned template-based approaches is that many of the patches in their search space are incorrect. To address this issue, constraint-based approaches try to directly generate a patch that is guaranteed to fix the bug. Consequently, these approaches explore a small, yet fruitful search space. To guarantee the plausibility of generated patches, constraint-based APR formulates the test input/output requirements as constraint specifications [47]. Then, solvers are used to find the solution for the specified constraints and a patch is generated incorporating the found solution.

Hoang et al. [83] propose SemFix, an APR tool for fixing *single-line* bugs. SemFix employs a simple fix template by changing a single statement. SemFix uses symbolic execution to compute the constraints that should be satisfied for passing the tests. It then uses satisfiability modulo theories (SMT) solvers to find the solution for the extracted constraints. Later DirectFix [84] and Angelix [85] build upon SemFix to fix multi-line bugs more efficiently. For this, Angelix uses a simple and lightweight constraint, called angelic forest. The essential quality of angelic forest constraints is that their size is independent of the size of the program. This makes the constraint solving problem significantly more efficient and as a result Angelix achieves an effectiveness comparable to GenProg. Afzal et al. [86] introduce SOSRepair. SOSRepair first uses symbolic execution to find code components semantically similar to the buggy code. Next, it uses constraint solving on the similar code to generate patches that pass all the tests. SearchRepair [87] also encodes a codebase as constraints on input-outputs. Then, it encodes the buggy code as constraints and finally, uses SMT solvers to find code components from the codebase that solve the buggy code constraints.

Xuan et al. [14] introduce Nopol. Nopol uses a fix template that modifies if conditions to repair the code. For this, Nopol finds angelic values that make the tests pass. Based on the identified angelic values, Nopol builds constraints that should hold to pass the tests. Then, SMT solvers are used to find solutions for the extracted constraints. The result is a plausible patch with a new expression for the if condition. Durieux and Monperrus [88] build upon Nopol by adding method calls to the expressions that can replace faulty if conditions. Long and Rinard [57] propose staged program repair (SPR) which uses a constraint-based approach in three steps. First, it applies parameterized templates to build a large search space of possible patches. Next, it searches for target values, especially for logical condition, that should replace the template parameters to pass the tests. Finally, solvers are used to find concrete expressions to replace the logical conditions and achieve the target values.

Template-based APR for Static Analyzer Issues: Template-based APR is also employed for fixing issues detected by static analyzers. Bader et al. [89] introduce an approach, called Getafix, to fix bugs detected by two static analyzer tools, Infer [90] and Error Prone [91]. First, they employ a hierarchical clustering algorithm to extract a hierarchy of fix patterns from general to specific ones from their training dataset. Next, they use these fix patterns to make fix suggestions. Finally, they rank the fix suggestions and recommend the best ones to the developer.

Marcilio et al.'s [92] introduce SPONGEBUGS which is a program transformation technique that fixes violations of 11 SONARJAVA rules². SPONGEBUGS applies predefined transformation templates to repair SONARJAVA warnings. The templates adopted by SPONGEBUGS identify the warnings and modify the

²<https://github.com/SonarSource/sonar-java>

code at textual level. The search space of SPONGEBUGS contains just one single patch, the patch produced by the relevant template. This makes SPONGEBUGS an efficient APR approach. The main limitation of SPONGEBUGS is that it needs a carefully designed template for fixing violations of each SONARJAVA rule. The design of such templates is complicated and time-consuming.

Summary: The template-based APR is an effective repair approach; methods like TBar [82] are still considered for comparison against most advanced techniques [93]. However, these techniques usually have a very large search space that are hard to explore given the limited resources in real-world projects. Constraint-based APR solves the issue of too large search space by generating a patch that passes all tests by construct. The main limitation with the efficiency of constraint-based APR is the large resources needed for finding a solution for the extracted constraints. Approaches like Nopol [14] aim to mitigate this issue by finding angelic values instead of using symbolic execution. However, finding a solution for the constraints is still a notoriously time-consuming task [94], which makes the patch generation inefficient.

2.2.2 Learning-based Patch Generation

In recent years, the most advanced APR approaches employ learning-based techniques to predict the patch that correctly fixes the bug [71]. These approaches are based on a reasonable assumption: by looking at the bug fixing patterns in a large corpus of historical data, we can predict the patch that fixes a given bug. Note that we consider the learning-based approaches separate from the template-based approaches that use neural networks to extract their templates, such as AVATAR [80]. In contrast with AVATAR, learning-based approaches directly generate the patches, including the fix ingredients and specific tokens used in the patches.

Existing learning-based APR approaches employ neural networks models for patch generation. The model takes a buggy code as input and generates patches aimed at fixing the bug. This patch generation process is called *inference*. A learning-based APR approach can be categorized from four different aspects as follows:

- *What input is given to the model?* For example, the input to the model can be only the buggy code, or also include the relevant comments and the context around the buggy code.
- *In what format is the input given to the mode?* The input representation plays a significant role in the approach's performance [6]. For example, the buggy code can be represented as a sequence of tokens [95] or an AST [96].
- *How is the model built?* The APR approach can build a model from scratch [95, 97, 98], use an existing pre-trained model [93], or fine-tune a pre-trained model [99].

- *What configurations are used for inference?* Due to the statistical nature of neural networks, the model can generate multiple output patches. The APR approach can use various techniques, such as beam-search [100], to guide the number and prioritization of patches generated at the inference phase.

We now review the significant existing learning-based APR approaches with a focus on their efficiency and their answers to these four questions.

The early learning-based APR approaches formulate the repair process as a neural machine translation task (NMT). Tufano et al. [100] train a neural model on 787,178 bug-fixing commits collected from GitHub repositories. The original version in each commit is a buggy code and the patched version is the fixed code. The objective of the model is to “translate” a buggy method to the corresponding fixed method. They use a recurrent neural network (RNN) encoder-decoder architecture [101] for their model. The encoder takes the buggy method as a sequence of tokens and produces hidden states. The decoder takes the hidden states and generates the fixed method as a new sequence of tokens. The experiments by Tufano et al. [100] “indicate that NMT is a viable approach for learning how to fix code”.

Chen et al. [95] also look at APR as an NMT task. They propose SequenceR, which trains an encoder-decoder model to fix Java buggy methods. SequenceR particularly addressed one of the major challenges for learning-based APR methods: the unlimited vocabulary problem in big code. Note that the correct fix for a bug may contain any specific identifier or literal that is not present in the vocabulary of the training dataset. To deal with this issue, SequenceR employs a copy mechanism to copy tokens from the buggy version to the generated code. With this technique, an identifier or literal that appears in the buggy code can be used in the fixed version even if it does not exist in the vocabulary of training dataset. In addition to the buggy method, SequenceR also provides the class declaration, signature of other methods, and class fields in its input to the model. This makes the model aware of the bug context that can hint at the correct fix. Finally, SequenceR puts special markers before and after the buggy part of the input to further clarify which part of the code the model should modify. SequenceR sets a strong baseline for all later learning-based APR methods by fixing 14 of 75 single-line bugs in Defects4J.

White et al. [102] propose DeepRepair. DeepRepair utilizes two operators “add statement” and “replace statement” as its repair templates. For each of these templates, a new statement should be used as a fix ingredient. DeepRepair employs a deep neural network to compute the code similarity between the faulty statement and other statements in the codebase. The more similar statements are ranked higher in the list of potential fix ingredients. DeepRepair also modifies the statements selected as fix ingredients by modifying replacing their variables with variables in the scope. The experiments show that DeepRepair’s repair effectiveness is comparable to GenProg, while DeepRepair is more efficient by discovering compilable fix ingredients faster.

To improve learning-based APR effectiveness, researchers have incorporated the structure of the program into the model’s input. CoCoNut [103] uses convolutional neural network (CNN) instead of RNNs to capture hierarchical features of the program. Moreover, it splits the input by white spaces as well as numbers, camel letters, and underscores to achieve PL-specific tokenization. Li et al. [104] introduce DLFix, which utilizes a tree-based recurrent neural network for its model. The input to DLFix’ model is the AST of the buggy code, which also contains the structural information. Tang et al. [105] also utilize a tree-based model that encodes both the original sequence of input tokens and grammar-rules linking those tokens. They restrict the inference of their model to only generate grammar abiding token sequences to avoid syntax errors in their patches. Ye et al. [106] propose RewardRepair, which trains the model to generate error free patches. For this purpose, RewardRepair incorporates compilation and test execution errors into the loss function.

CODIT [107] is another learning-based APR approach that uses tree-based neural networks to learn valid code change patterns. CODIT trains the neural network from scratch on 24K real-world code changes and fixes 25 out of 80 considered bugs from Defects4J. Zhu et al. [108] introduce Recoder, which ensures the correctness of the syntax of the generated output. For this, Recoder uses a novel provider/decider architecture for the model’s decoder layer. Recoder is the first learning-based APR tool that outperforms other approaches, especially TBar [82], by fixing 53 bugs from Defects4J. Hoppity [96] treats the AST as a graph and adopts a gated graph neural network (GGNN) to learn bug-fixing patterns. In Hoppity, the GGNN predicts the correct AST node positions and edits on them. These works [96, 103, 104, 105, 106, 107, 108] make the patch generation process more efficient by generating fewer erroneous patches.

Jiang et al. [109] make a major step forward in learning-based APR by proposing CURE. CURE works in five main steps. First, it uses byte-pair encoding (BPE) to tokenize the inputs. This encoding splits the words into smaller sub-words with only a few characters. Consequently, the vocabulary size is small, which mitigates the out-of-vocabulary issue and leads to a smaller search space. In the second step, CURE trains a generative pre-trained transformer (GPT) [110] model on a large corpus of source code. This model encodes the structure of programming language texts and enables CURE to generate syntactically correct patches. Thirdly, CURE fine-tunes its pre-trained model on a dataset of bug-fix pairs extracted from the history of open-source projects. This enables the model to perform the APR task. Fourth, CURE investigates a given buggy program to find valid identifiers that could be used for replacing the faulty code. Finally, a new code-aware beam search strategy is used for inferring the bug-fixing patches. This new beam search strategy immediately discards the patches that contain a syntax error. It also penalizes the patches that are not similar to the original faulty statement. This leads to an efficient formation and exploration of the search space during patch generation.

Large Language Model (LLM) Based APR: The strong performance of CURE illustrates the repair power of neural networks trained on large corpus of data. Based on this observation, APR researchers have adopted LLMs that are already pre-trained on large corpus of source code and natural language[111]. The LLMs use Transformers [112] generally categorized into three groups: *encoder only*, *decoder only*, *encoder-decoder*. BERT [113] is an encoder only model that learns a representation of the data and is trained with a masked language model (MLM) objective. BERT predicts the masked part of its input. GPT [114] is a decoder only model, which iteratively predicts the next token based on the given input and previously predicted tokens. Finally, T5 [115] is an encoder-decoder model that uses its encoder to extract a proper representation of the input and then employs the decoder to generate the output from the extracted input representation. Existing LLM-based APR approaches use various types of LLMs as follows.

Xia and Zhang [116] introduce AlphaRepair, which directly employs CodeBERT [117] to fix bugs in various programming languages, such as Python and Java. Given a buggy code, AlphaRepair utilizes CodeBERT to generate a ranked list of k patches. First, AlphaRepair runs state-of-the-art FL techniques to identify the faulty statement. Next, it masks the faulty statement by replacing it with the multiple infill special token. The original faulty statement is added to the end of the buggy code as a comment. Next, this new version of the buggy code is given to the model as prompt³. AlphaRepair invokes the LLM once per each mask token. After the n th invocation, AlphaRepair keeps the list of k best sequence of candidate tokens that can replace n first mask tokens. At the end, AlphaRepair generates k patches that CodeBERT considers to have the highest likelihood of being correct. This method of searching for the best candidate patches contributes to the efficiency of AlphaRepair. The experiments by Xia and Zhang show that AlphaRepair outperforms all existing state-of-the-art APR approaches a the time by fixing 3.3X more bugs from Defects4J compared to the baselines.

CIRCLE [118] takes T5 as its base LLM and fine-tunes it with continual learning. CIRCLE provides the buggy line and its context in its prompt and asks for the fixed code with a mask token. Listing 2.2 shows an example CIRCLE prompt. The buggy line is presented at line 1, the context comes at lines 2-8 and the fixed code is asked for with a “[P]” token at line 9. With a comparison against Recoder [108], TBar [82], CURE [109], and FixMiner [79], the authors find that these methods are complementary. This means at time of developing CIRCLE, the LLM-based APR approaches were still not strong enough to completely replace more traditional approaches, such as TBar.

With the release of OpenAI’s powerful GPT models [119], more attention is given to LLM-based APR and the approaches are improving fast. Xia et al. [93] conduct a large experiment to study the APR effectiveness of state-of-

³In the context of LLMs, the input to the model is also called the *prompt*.

```

1 Buggyline: n ^= n - 1
2 Context:
3 def bitcount(n):
4     count = 0
5     while n:
6         n ^= n - 1
7         count += 1
8     return count
9 The fixed code is: [P]

```

Listing 2.2: An example prompt used by CIRCLE [118].

the-art LLMs. To fix a given bug, they use a few-shot prompt that contains two example `<buggy_code,fixed_code>` pairs. The first example is a simple Fibonacci bug and the second one is a small bug from the project of the given bug. They consider nine LLMs and compare their performance with AlphaRepair, RewardRepair, Recoder, TBar, and CURE. They find that Codex, a model by OpenAI built on top of GPT-3 and trained with a large corpus of source code, outperforms all other models and tools by fixing 99 Defects4J-1.2 bugs. Notably, they use `temperature=0.8` and `sample_size=200` as settings for invoking Codex. This means they ask the model to generate 200 responses at each invocation and allow it to add randomness to its responses as the temperature is higher than zero. In 2022, this study made it clear that LLMs are able to significantly outperform previous APR approaches and many program repair researchers shifted towards using LLMs, accordingly.

Note that using a `sample_size=n` and `temperature=t` for generating n multiple patches is similar to the method used by AlphaRepair and different from beam search [120]. Beam search is a deterministic method that produces the n patches with the highest likelihood of being the correct answer per model’s trained parameters. Contrarily, in sampling, the patches are generated token by token. At step i , n the token sequences from step $i - 1$ are taken and n extensions of them with highest likelihood of being correct are computed. This makes the sampling strategy nondeterministic and the temperature value notes how much randomness the model is allowed to consider for patch generation. Overall, beam search yields better results, while sampling is more scalable.

Researchers have improved the effectiveness of APR approaches with prompt engineering [121] and fine-tuning [122]. Zhang et al. [123] use infill prompts for APR and mask the tokens based on common template-based APR patterns. Cao et al. [124] show that adding the code intention to the prompt and using a dialogue based approach improves the LLM’s performance. Fakhoury et al. introduce *nl2fix*, which augments the prompt with issue descriptions related to the bug. Nashit et al. [125] use an embedding-based technique, SROBERTa[126], and a frequency-based technique, BM-25[127], to retrieve relevant examples for a

given repair task. Similar to [93], they show how Codex with a few-shot prompt outperforms the state-of-the-art APR tools.

Ahmed and Devanbu propose using the self-consistency approach for program repair [128]. They use a few-shot prompt that contains the buggy code and the model is asked to generate multiple pairs of explanation-solutions for the bug. The most frequent solution is taken as model’s final answer. This approach outperforms the state-of-the-art on the MODIT dataset [129]. The prompt engineering strategies used by these techniques gives extra information to the LLM to make it look for patches that are more likely to fix the bug. This improves the efficiency of APR by not spending resource for generating many incorrect patches.

Fine-tuning is also used for enhancing LLM-based APR effectiveness [130, 131, 132]. Mashhadi and Hemmati [133] fine-tune CodeBERT to automatically generate fixes for ManySStuBs4J [134]. Zirak and Hemmati show that fine-tuning a LLMs on a dataset related to the application domain is significantly more effective than fine-tuning on an irrelevant dataset [135]. Jiang et al. [21] conduct a large-scale study on using LLMs for APR. They show a fine-tuned version of InCoder [136] that fixes 164% more bugs compared to the state-of-the-art learning-based techniques. In another study, Huang et al. [137] observe that fine-tuning is effective for fixing different types of bugs (test failure, vulnerability, and errors) in various programming languages (Java, C/C++, and JavaScript).

There are LLM-based APR approaches that adopt an iterative prompting strategy [118, 138, 139, 140, 141]. In this strategy, the repair process does not stop after receiving the first batch of patches from an LLM. Instead, some feedback is provided to the LLM and it is requested to generate new patches. Most recently, Xia and Zhang introduce ChatRepair [45]. ChatRepair gives the test results as feedback to the LLM to improve its patches. Once a plausible patch is generated, ChatRepair asks the LLM to generate alternative plausible patches. The iterative approach improves the efficiency by guiding the LLM through the exploration of possible patches with informative feedback.

The most recent advancement in APR, is the invention agent-based approaches [142]. In these approaches, the LLM is used not only for patch generation, but for different roles that developers play during debugging. STEAM utilizes ChatGPT [119] to report the bug, explain the bug, generate a patch, review the patch, and improve the patch iteratively. RepairAgent [143] uses the LLM to understand the bug, collect information for bug-fixing, generate a patch, and improve the patch iteratively. The agent-based approaches go beyond direct patch generation with LLMs and use LLMs for steps 1-4 of APR as described in Figure 2.1. Despite its effectiveness, the agent-based approach can be costly by generating too many tokens using the LLM.

Learning-based APR for Static Analyzer Issues: TFix [144] takes advantage of a learning-based approach to fix static warnings. It formulates fixing as a text-to-text task that uses neural networks to translate a rule violating pro-

gram to a violation free version. TFix specifically fixes violations of ESLINT rules in JavaScript programs. VRepair [145] and SeqTrans [146] propose an APR approach for fixing security vulnerabilities. They first pre-train a model using a large dataset of bug-fixing code changes. Next, they fine-tune their model with vulnerability fixing code changes. Fu et al. [147] introduce VulRepair, which employs a fine-tuned version of CodeT5 [148] to fix vulnerabilities.

Wadhwa et al. [149] introduce CORE. CORE fixes issues detected by CodeQL and SONARQUBE. CORE uses GPT-3.5-turbo to generate patches and employs GPT-4 to rank the patches that remove the issues. CORE’s prompt contains 1) the buggy program and its context, 2) the warning message by the static analyzer, and 3) the static analyzer’s documentation that explains the issue and the way to fix it. The experiments by the authors show that CORE outperforms template-based APR tools for fixing static analyzer issues.

Summary: As the most recent APR technique, LLM-based APR has shown promising effectiveness. Considering the costs of using LLMs [150], it is important to pay special attention to the efficiency of these approaches. In this regard, efficient methods should be designed to explore the search space and obtain a correct patch, while generating as few patches as possible.

2.3 Automated Patch Assessment

After the patch generation step, automated techniques are employed to detect *overfitting* patches: the patches that comply with the specification, but fail to correctly implement the intended behavior [151]. Automated overfitting detection makes APR more efficient by reducing the need for manual assessment, which is time-consuming and not perfectly accurate [152].

Test-suite Improvement: The overfitting issue can be alleviated by making the test-suite stronger, as a stronger test-suite invalidates more incorrect patches. Xin et al. propose DiffTGen [153], a tool that generates tests that generate new test cases that differentiate between the buggy and patched programs. These new tests indicate that the patch is changing a part of the buggy program behavior that is not captured by the original test. This could lead to regression issues that make the patch incorrect. Yu et al. [154] introduce a similar approach, called UnsatGuided. Yu et al. note that this approach only detects regression issues in the patch, while it does not detect incomplete fixes. In an incomplete fix, the wrong behavior of the buggy program is changed, however the new behavior still does not comply with the expectations. Xiong et al. [154] propose another approach that operates in two steps. This approach first generates new test cases. Next, it compares the behavior of the buggy and patch programs on all the original and new generated tests. A patch that behaves similar to the buggy program on passing tests and different from it on failing tests is more likely to

be correct. Dong et al. [155] build a tool, called ETPAT, that improves the test-suite particularly for expression modifying APR techniques. ETPAT generates new test cases that capture the differences between the original and patched expressions.

Ye et al. [15] propose Random Testing with Ground-truth (RTG) for overfitting detection. RTG assumes the ground-truth correct patch is given. It uses Randoop [156] and Evosuite [19] to generate new tests that detect the behavioral differences between the ground-truth and APR patches. The authors later conduct a large scale study on APR patches for QuixBugs bugs [157]. Their experiments show that RTG has a precision as high as 98.2% on this dataset. This indicates the strong capability of RTG, however this technique is not useful in practice as the ground-truth correct patch is not present.

Overfitting detection via test-suite improvement has two main limitations. First, test generation and execution is well-known to require significant resources that may not be available. Second, as the correct fix for the bug is not given, a perfect oracle is not available. This makes the test improvement techniques unable to detect incomplete fixes as mentioned by Yu et al. [154]. To address the lack of oracle, researchers have used fuzzers to produce new test inputs that obtain significant information about the patches without knowing the expected output. For example, Yang et al. and Gao et al. use generate test inputs to check if the patch crashes or causes memory-safety problems [158, 159]. Though effective, this method does not detect incorrect patches that successfully execute, while producing wrong outputs.

Machine-learning Classification: Researchers have extracted static [15] and dynamic [160] features from APR patches to classify them as correct or overfitting. Tan et al. [161] conduct a static analysis on code transformations to decide if the patch contains any anti-patterns. Patches with anti-patterns are discarded to make the search space smaller. Wang et al. [162] employ machine-learning (ML) techniques to predict the overfitting patches using eight statically extracted features. Lin et al. [162] propose Cache. Cache first extracts the changed code, its context, and the corresponding AST structure from the patch. Then, it uses a deep neural network to classify the patch as overfitting or correct.

LLMs are also employed for overfitting detection. Tian et al. [163] compute BERT embeddings for the patch and use a logistic regression to predict the patch correctness based on the embeddings. Zhang et al. [164] also use a fine-tuned version of BERT for overfitting detection. Yang et al. [165] measure the difference between the entropy of an LLM on the buggy code and its patched version. They call this difference value, entropy-delta and consider it as a measure of naturalness of the patch. Their experiments show the correctness of a patch can be predicted based on its naturalness. Le-Cong et al. [166] propose Invalidator, a two-step approach for overfitting detection. In the first step, Invalidator extracts invariants from the ground-truth and patched versions. The invariants are compared against

each other to check if the patch overfits. In the second step, an LLM is used to capture the syntactical difference between the two versions. Finally, an ML model is used to predict the patch correctness based on the extracted syntactical differences.

Given the difficulty of accurately predicting the correctness of patches, researchers have also investigated patch ranking and patch clustering as alternatives. A patch ranking technique prioritizes the patches for manual assessment; a patch with a higher rank is more likely to be correct. With this technique the developer is likely to manually assess fewer patches. Bhuiyan et al. [16] introduce PrevaRank. PrevaRank first studies the bug-fixing history of open-source projects to learn the correctness likelihood of various types of patches. The result of this first step is a probabilistic model that predicts patch correctness likelihood. Next, PrevaRank find the type of a APR patches ranks them based on their correctness likelihood.

Martinez et al. [9] automatically cluster the generated patches to minimize the manual patch assessment effort. They use Randoop and EvoSuite to generate tests for plausible patches. Then the patches are clustered based on their test results (fail/pass): all patches with same result are clustered into the same group. By rejecting one patch from a cluster, all the patches in that cluster are eliminated. This method reduces the number of patches to be manually analyzed by 50%.

2.4 Manual Patch Assessment

The patches that pass the automated patch assessment step should then be manually analyzed to determine their correctness. This manual analysis step is closely related to the code review problem, which is widely studied [167]. As reviewing code diffs is notoriously hard [18], various techniques are suggested to help developers in this regard.

Natural Language Patch Explanation: A common technique for helping developers in assessing code diffs is explaining the patch in natural language. Marwan et al. [168] show that accompanying a code hint by an explanation, makes users more likely to use the hint. A proper patch explanation should describe *what* is changed in the code and *why* it is changed [169].

There are studies using predefined rules or templates for automatic patch explanation generation. Buse and Weimer introduced DeltaDoc[169]. DeltaDoc produces summaries that are longer than normal commit messages but shorter than raw diffs. This approach consists of three steps. In the first step, for each statement, the conditions under which that statement can be executed are extracted. In the second step, for each old statement like Z that would be executed under a condition like X and is changed to a new statement like Y, a document of the form “When calling A(), If X, Do Y Instead of Z” is generated. Finally, he

generated document is summarized using several heuristics. The *ChangeScribe* tool [170] and the technique by Shen et al. [171] both use method stereotypes distribution [172] in a patch to determine the patch type. The determined type clarifies “*why* the source code is changed in this commit?” The method presented by Shen et al. [171] also identifies the maintenance type of the change (ex., corrective, perfective) and reports it in the generated explanation.

Another trend in the patch explanation generation research field is using NMT to translate the code diff into natural language explanation [173, 174, 175, 176]. Jiang et al. [177] propose using a RNN Encoder-Decoder architecture for the translation. In [178], we show that this technique does not fully take advantage of cross-project learning for explanation generation. PtrGNCMsg [179] is another tool that addresses the out-of-vocabulary (OOV) problem by using pointer-generator network. In this manner, before generating each of the explanation tokens, PtrGNCMsg uses probabilistic techniques to decide whether the word should be selected from the training vocabulary or copied from the given test commit diff.

CODISUM [180] and ATOM [181] are commit message generation methods that also take the code change structural information into account to generate the messages. In this regard, CODISUM [180] replaces class/method/variable names with placeholders to extract the structural information of source code changes. ATOM [181] sees the code change diffs as changes in the abstract syntax trees (AST) of programs instead of plain text. ATOM [181] also presents a retrieval technique which finds the most relevant training diff to a given test diff in terms of the tf-idf score [182]. Finally, it ranks the generated and retrieved messages and returns the better one as the output. [183] trains a dual-objective natural translation model for commit message generation. The model aims to generate a description that is close to the ground-truth description and also correctly predicts the cluster of the patch. The patches are clustered according to their ground-truth descriptions and the generated description should be able to predict the cluster correctly.

Liu et al. [184] propose a method to automatically generate pull request descriptions from commit messages and added code comments in a pull request. Their proposed approach uses an attentional encoder-decoder architecture. Fang et al. [185] also translate commit messages and code comments of a pull request to PR descriptions. Their tool, called PRHAN, employs a hybrid attention mechanism to improve the performance and byte pair encoding to alleviate the out-of-vocabulary problem.

With the advent of LLMs, researchers have leveraged these models for patch explanation generation as well. Mahbub et al. propose Bugsplainer [186]. Bugsplainer use CodeT5 architecture to train a model that takes the buggy and patched code and generates a commit message. The code is presented to the model in *diffSBT* format, which includes the changed AST nodes with its structural information. Sobania et al. [17] use GPT-3.5 to explain Arja [23] patches for 30 bugs. They ask the model to explain the condition, cause, position, consequence, and change of

the patch. They find the explanations are mostly correct but in many cases lack completeness. They also conclude that the explanations for machine generated patches are better than explanations for human patches.

Improving Code Diff Presentation: We can also aid code reviewing by augmenting the code diff with other types of information or presenting it more efficiently. This can be done in different ways, such as splitting code changes into smaller sets of related changes [187] or by improving how code changes are presented. Some tools improve the code diff presentation by making it more fine-grained, such as MERGELY [188] and GUMTREE [189]. For this, MERGELY considers changes at the level of code elements inside lines, instead of looking at whole line differences. GUMTREE enhances existing AST differencing algorithms, such as CHANGEDISTILLER [190], and proposes an algorithm that computes minimum changes that should be made on the original AST to reach the patched one. GUMTREE marks these fine-grained AST level changes on the code diff to help developers easily see what is the exact change to the program. Decker et al. [191] introduce a new syntactic differencing tool, called SRCDIFF. Based on manual and statistical analysis, they propose several heuristics to determine if a change is actually a modification or a complete replacement. In contrast with GUMTREE, the goal of SRCDIFF is not producing an optimal diff. SRCDIFF intends to produce a diff that is more similar to the changes performed by developers.

In another line of code, researchers try to integrate new information into the integrated development environments (IDE) [192]. This can include putting code change information with data related to developers' actions in the IDE [193] or production data [194]. For example, Cito et al. [195] add the time spent to run a method at production to the IDE code editor. When the code is changed, they predict the new production time for the changed code and show it to the developer in IDE. As a result, developers better detect performance problems in their code changes. Bohnet et al. [196] combine execution traces with code changes to help developers identify the root cause of a failure in C/C++ programs. They define the execution trace as the sequence of executed functions. When a program faces a failure after a code change, the execution trace is collected and shown to the developer. Next, the developer interacts with the tool to detect the failure causing behavioral change.

Kanda et al. [197] propose Didiff. Didiff runs two versions of a program and illustrates the difference between the values assigned to program variables in a web interface. Didiff uses SeLogger [198] to record all the values each variable of a program holds during an execution. Using these values, Didiff creates two lists of values for each variable access that is not part of the code change. The first list represents the values assigned to the variable at that point of access in the original version. The second list represents values assigned to the variable in the patched version of the program. Finally, Didiff compares the two lists of values for each variable access. The result of this comparison is presented to code

reviewers in form of a graphical user interface (GUI). This GUI shows the two lists of values assigned to a variable and highlights if there is a difference between these two lists.

Liang et al. [18] design an interactive tool, called InPaFer, for APR patch assessment. InPaFer collects runtime data, such as variable values, execution trace, while executing the tests on generated patches. It then asks the code reviewer about the correctness of runtime behavior of a patch. If the code reviewer labels the behavior as incorrect, that patch and similar patches are marked as incorrect and eliminated. The authors show that InPaFer reduces the code reviewing time and increases the number of repaired patches.

To make the patch assessment more convenient for developers, the APR patch suggestion should be integrated into existing development environments. In this regard, Urli et al. [199, 200] introduce Repairnator, as a program repair bot. Repairnator constantly monitors Travis-CI builds on GitHub repositories. When a build fails, Repairnator reproduces the failure locally, uses repair tools to patch the program, finds a plausible patch that passes all the tests, and submits the passing patch as a pull request to maintainers of the failing project. SapFix is a program repair bot introduced by Marginean et al. [201] at Facebook. SapFix is integrated into the Phabricator continuous integration platform. Once a new diff is submitted on Phabricator, SapFix employs three strategies: mutation-based, template-based, and revert-based (which just reverts to the old version) changes to generate a set of patches that pass all tests. Next, the patches that pass all the tests are prioritized, and the top one is reported to developers as a fix.

C-3PR [202] monitors pushes on Git repositories and looks for changed files with static warnings. Next, it applies appropriate transformations to fix the warnings. If a warning is fixed by transformations, C-3PR submits a pull request including the fix. Serban et al. [203] introduce SAW-BOT, which fixes violations of five SONARJAVA code smell rules. The authors compare three ways of suggesting fixes. First, the legacy mode in which all violations in the whole project are fixed and the fixes are suggested in a pull request. Second, the pull request mode in which only violations in a recently created pull request are fixed and then suggested in a separate pull request. Third, only violations in a pull request are fixed and suggested as part of a code review. The authors conclude that developers prefer the third option because they can focus on reviewing the soundness of the fix rather than on understanding why the fix is there.

3

Thesis Contributions

In this section, we present how we address the problems stated in section 1.2 and make patch generation, automated patch assessment and manual patch assessment efficient.

3.1 P1: Efficiently Guiding Patch Generation Towards Likely Correct Patches

As explained in section 2.2, the efficiency of APR faces a major challenge in form of a trade-off. On the one hand, a group of approaches, like Cardumen [13], employ a simple technique to generate a large set of possible novel code components. This puts the heavy burden of checking patch plausibility and correctness on the patch validation and patch assessment steps. On the other hand, other approaches, such as Nopol [14], aim to generate novel code components that are guaranteed or likely to form a plausible or correct patch. Seeking such guarantees, for example by finding a solution for constraints, usually require large computing resources. In summary, to make APR efficient, we need guide patch generation toward synthesizing patches that are likely to be correct, while using minimal computing resources.

We now present two contributions we have made to address the challenge to patch generation efficiency. The first contribution is a highly precise template-based APR approach, called SORALD for fixing static analyzer issues. The second contribution proposes an efficient prompting strategy for LLM-based APR that is implemented in CIGAR. CIGAR guides the LLM towards generating correct patches using minimal number of tokens.

```

1 Runnable runnable = () -> System.out.
  println("Hello_World");
2 Thread myThread = new Thread(runnable);
3 myThread.run();
4 myThread.start();

```

Listing 3.1: A correct patch generated by SORALD for a violation of SONARJAVA’s rule *S1217*.



Figure 3.1: Official SONARJAVA documentation for rule *S1217*.

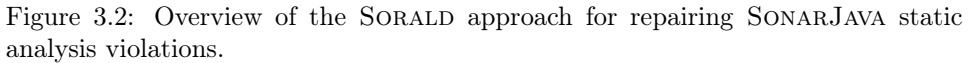
3.1.1 Contribution 1: A Highly Precise Template-based Approach for Fixing Static Analyzer Issues

Template-based APR approaches employ fix templates that tend to generate likely correct patches. This makes design of such valuable fix templates essential for APR efficiency. Researchers have employed domain knowledge of developers [28] and historical bug-fixing data [79] to create accurate fix templates. Even with the notable effort dedicated to fix template design, template-based approaches are not guaranteed to fix a bug for three reasons. First, it is not clear which template should be applied when multiple templates are applicable. Second, to finalize a patch, a template may need to copy fix ingredients from the existing code. Which fix ingredient should be used is not clear. Finally, many bugs do not follow a common structure. A template-based approach may allocate resources for fixing such bugs, while no manually or automatically crafted template can fix them. These three challenges make template-based APR imprecise and negatively impact its efficiency.

To make a precise template-based APR, we propose SORALD: a novel template-based APR approach for fixing SONARJAVA issues. SORALD relies on SONARJAVA’s official documentation for designing accurate fix templates that lead to a precise search space. SONARJAVA is a static analyzer that defines 631 rules and statically analyzes a given Java program to find violations of those rules. A violation is reported as an issue and indicates a potential functional bug, code smell, or vulnerability. SONARJAVA also provides an official documentation¹ that describes each rule and how its violations should be fixed. SORALD contains fix templates that modify the AST of a rule violating program to eliminate SONARJAVA issues. The employed fix templates are based on SONARJAVA’s official documentation to ensure the generated patches are correct. Consequently, SORALD has an efficient patch generation method that directly synthesizes the patch that is guaranteed by the documentation to be correct.

For example, Figure 3.1 shows the header of SONARJAVA’s documentation for its rule *S1217*. This rule dictates that the method `Thread.run()` should not be called directly. The suggested fix for violations of this rule is replacing `Thread.run()` with `Thread.start()`. Based on this suggestion, SORALD has a

¹<https://rules.sonarsource.com/java/>



SORALD Approach

At the heart of SORALD, there is a fix template for each SONARJAVA rule that determines how the AST of a rule violating program should be transformed to eliminate the violation. The fix templates employed by SORALD do not require fix ingredient extraction. This makes the fix templates deterministic, they yield exactly one patch. SORALD employs the SPOON [22] meta-programming tool to obtain the AST of subject programs and transform them. The technically sound

design of SORALD makes it possible to encode a fix template with a few dozens line of code. Today, SORALD fixes violations of 30 SONARJAVA rules, which indicates its extensibility. Also, the pretty-printer implemented in SORALD ensures the code diff is presented with minimal white space differences to make the code review smooth. The pretty-printer together with the integration into GitHub, provided by SORALDBOT, facilitates the manual assessment step.

Evaluation of SORALD Effectiveness

To evaluate the effectiveness of SORALD, we run it on popular open-source Java project from GitHub. In this experiment, we assess SORALD’s effectiveness and efficiency in fixing SONARJAVA reported issues. We specifically use SORALD to fix violations of 10 SONARJAVA rules that are labeled as potential functional “bug”. We focus on “bug” rules as they are usually seen as more important by developers. For each rule, we run SORALD once to fix all *target violations* of that rule. We also check if SORALD code transformations keep the existing intended behavior of the program, which is essential for generating correct patches.

Dataset: For this experiment, we select the GitHub projects that meet the following requirements:

1. Popular project: have at least 50 stars on GitHub.
2. Maven project: have a “pom.xml” file in the root directory.
3. Active project: have at least one commit in the past three months as of Nov 2020.
4. Pull request friendly project: have a pull request accepted in the last three months as of Nov 2020.
5. Healthy project: pass “mvn compile” and “mvn test” commands successfully with OpenJDK Java 11 on the last commit in Nov 2020.
6. Continuous integration friendly project: the project uses CI, which is checked as having a “.travis.yml” in the root directory.

The result of these filters is a set of 161 GitHub projects. SORALD is applicable on these projects and we can expect to receive their feedback on SORALD fixes.

Evaluation Metrics: For each of the 10 considered rules, we compute the number of violations detected by SONARJAVA (“DV”), the number of target violations (“TV”), and the number of violations fixed by SORALD (“FV”). In particular, we are interested in the ratio of fixed violations to target violations, which indicates the effectiveness of SORALD. Moreover, we compute the number of projects on which SORALD code transformations cause test failure (“Failing_Repos”). A low

Table 3.1: The effectiveness and efficiency of SORALD on 161 open source projects. SORALD successfully repairs a majority of the target violations (65%) in the order of magnitude of seconds per violation, which saves valuable time of human developers.

SQID	TDR (TV/DV)	FTR (FV/TV)	Failing_Repos	MT (sec)
S1217	100% (2/2)	100% (2/2)	0.0% (0/161)	4.5
S1860	100% (5/5)	100% (5/5)	0.0% (0/161)	4.4
S2095	46% (361/782)	9% (34/361)	0.6% (1/161)	6.3
S2111	100% (69/69)	53% (37/69)	1.2% (2/161)	4.9
S2116	100% (1/1)	100% (1/1)	0.0% (0/161)	4.5
S2142	99% (315/316)	95% (300/315)	0.0% (0/161)	4.6
S2184	97% (431/440)	85% (368/431)	0.6% (1/161)	4.5
S2225	13% (3/22)	100% (3/3)	0.6% (1/161)	4.5
S2272	97% (40/41)	85% (34/40)	0.0% (0/161)	4.5
S4973	98% (80/81)	85% (68/80)	1.2% (2/161)	4.4
ALL	74% (1,307/1,759)	65% (852/1,307)	0.4% (7/1,610)	–

^a Columns “TV”, “DV”, and “FV” indicate the numbers of target violations, all detected violations, and fixed violations for each rule. Column “TDR” shows the percentage of detected violations that are targeted. Column “FTR” gives the ratio of fixed target violations. Column “Failing_Repos” shows on how many projects there are tests failing because of SORALD. Column “MT” notes the median time spent by SORALD to fix violations of that rule in a single project.

number of failures indicates that SORALD keeps the existing intended behavior of the program. Finally, we also measure the median time SORALD takes to fix violations of each rule on a project (“MT”). This time shows how efficiently SORALD works.

Experimental Results: Table 3.1 presents the results of this experiment, which is aimed at assessing the effectiveness and efficiency of SORALD. In this table, “SQID” is the identifier of SONARJAVA rules. “TV” and “DV” represent the number of target violations and all detected violations, respectively. The “TDR” column shows what percentage of detected violations are target. Also, “FV” shows the number of *fixed violations* for each rule. Based on these numbers, we compute “FTR”, the ratio of fixed target violations. The “Failing_Repos” column shows on how many projects (out of 161) there are tests failing because of SORALD. Finally, for each rule, “MT” notes the median time (in seconds) spent by SORALD to fix violations of that rule in a single project.

The first column indicates the dataset contains 1,759 violations of the ten considered rules. This shows the dataset is large and the experiment draws meaningful conclusions. This column also notes that 74% (1,307/1,759) of the all violations are considered as target by SORALD, which indicates its applicability on this large dataset.

The second column (“FTR”) shows that 65% (852/1,307) of the target violations are fixed by SORALD. This indicates that SORALD is an effective APR tool for fixing SONARJAVA issues. It is important to note that SORALD fixes each

Table 3.2: Comprehensive Manual Analysis of the 153 SONARJAVA “Bug” Rules.

Rule Type	#Rules	Prevalent in real-world?	
		Yes	No
Bug rules	153 (100%)	–	–
Fully fixable rules	77 (50%)	27 (18%)	50 (32%)
Partially fixable rules	20 (13%)	9 (6%)	11 (7%)
Unfixable rules	56 (37%)	17 (12%)	39 (25%)

of these violations by generating only one patch. This shows the fix templates employed by SORALD are highly precise by fixing 65% of violations with a single patch. The “Failing_Repos” column also indicates that most of SORALD transformations keep the intended part of program behavior. Only on 0.4% (7/1,610) of its executions, SORALD introduces a test failure. Finally, the “MT” column shows that SORALD’s efficient APR strategy leads to a fast performance as well. On median, it takes between 4-7 seconds for SORALD to fix all violations of a rule on each repository.

We also conduct another experiment to assess developers’ feedback to SORALDBOT pull requests. The PRs suggest SORALD fixes generated in our first experiment on 161 GitHub projects. To ensure the developers are interested to review SORALDBOT PRs, we only submit SORALD patches that fix a rule violation that is introduced in a late commit. In total, we submit 29 PRs to 21 GitHub projects. Among these PRs, 17 of them are accepted, 10 of them are rejected, and two are pending. Overall, in this experiment we receive positive feedback from developers that shows SORALDBOT properly facilitates manual assessment of APR patches.

Limitation of SORALD: As a template-based approach, SORALD has two main limitations. First, to fix violations of each SONARJAVA rule, a new template should be manually crafted. Designing such templates requires human expertise and effort that is not easily achievable. Secondly, there are rule violations that can only be fixed by understanding the context of the violation. As described in section 3.1.1, SORALD fix templates are deterministic, they generate exactly one patch per violation. Consequently, these templates do not take into account the context of violations. As a result of this limitation, SORALD is able to fix the violations of only a subset of SONARJAVA rules.

We conduct a manual analysis of SONARJAVA “bug” rules to assess on how many of the rules SORALD’s approach is applicable. We divide the rules into three groups: 1) Fully fixable rules: all violations of these rules can be fixed by a deterministic SORALD template according the SONARJAVA documentation; 2) Partially fixable rules: a subset of violations of these rules can be fixed by a SORALD template; and 3) Unfixable rules: it is impossible to design a SORALD template that fixes any of the violation of these rules with a guarantee.

Table 3.2 presents the result of this manual analysis. There are 153 “bug” rules in total. As the table shows, 50%, 13%, and 36% of these rules are fully fixable, partially fixable, and unfixable, respectively. These results indicate that for half of the rules, we cannot design templates that fix all their violations with a guarantee. We also assess if violations of these rules exist in our dataset of 161 GitHub projects. The “Prevalent in real-world?” column of Table 3.2 presents the results of this assessment. We see that violations of each group of rules are in fact prevalent in real-world as well. This indicates the importance of addressing the limitation of SORALD on partially fixable and unfixable rules.

One of the main strengths of learning-based methods is being flexible and considering the context of bug for patch generation. This significantly help in addressing the limitation of SORALD’s template-based approach. Therefore, we propose a state-of-the-art learning-based approach with an efficient patch generation technique in our second contribution as follows.

3.1.2 Contribution 2: An LLM-based APR Approach with a Highly Efficient Search Space Exploration Strategy

Most advanced LLM-based APR approaches take advantage of LLMs’ ability to generate many various patches based on the bug context [93]. This makes these approaches strong as their vast and context-relevant search space is likely to contain a correct patch. However, this also poses a challenge to their efficiency: exploring the large search space requires significant resources. Notably the state-of-the-art LLMs, such as OPENAI models, charge their users per each token that is either sent to them or generated by them.

To address the challenge to LLM-based APR efficiency, we introduce CIGAR, the first LLM-based APR tool that aims at minimizing the search space exploration cost, as measured by the number of tokens employed. CIGAR achieves cost-effectiveness with the help of its three delicately designed prompts working in concert: an ‘initiation prompt’, an ‘improvement prompt’, and a ‘multiplication prompt’. The initiation prompt initializes the repair process. The improvement prompt improves partial patches until a plausible patch is generated, avoiding throwing away potentially useful patches, hence avoiding wasting tokens. Finally, the multiplication prompt builds upon the already generated plausible patches to synthesize more plausible patches with diversity maximization. All these prompts are designed to be concise while staying informative, minimizing the overall token cost. The prompts help the LLM avoid unnecessary token cost by building upon its previous responses. CIGAR also uses a reboot strategy to allow the model to look into various parts of its search space, instead of spending tokens in dead-ends. In short, CIGAR enables the LLM to find diverse plausible patches by: 1) summarizing the feedback given to the LLM as a part of an iterative approach; 2) rebooting the repair process after a few failing LLM invocations to allow the LLM look at various parts of its search space; 3) employing the LLM to multiply the already generated patches in order to maximum diversity.

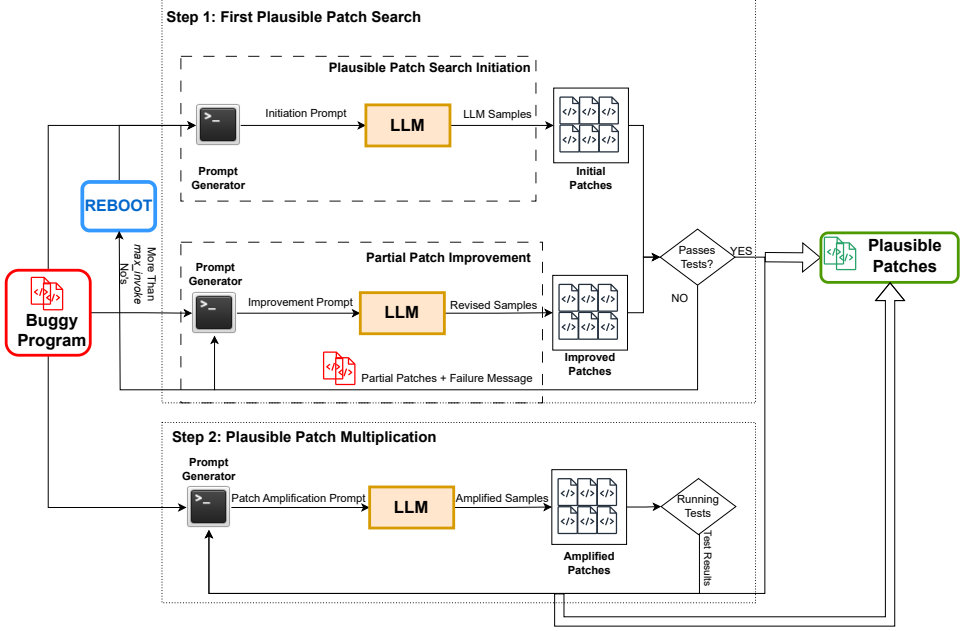


Figure 3.3: Overview of CIGAR: CIGAR combines original prompting techniques, incl. partial patch improvement, search rebooting and patch multiplication to reach high effectiveness at a low token cost.

CIGAR Approach

Figure 3.3 represents an overview of how CIGAR works. CIGAR takes as input a buggy Java function that fails on a test and generates a set of plausible patches that pass all tests as the output. Note that CIGAR fixes single-function bugs, bugs that can be fixed by modifying just one function. CIGAR works in two major steps, “First Plausible Patch Search” and “Plausible Patch Multiplication”. In the first plausible patch search step, CIGAR tries to generate a plausible patch that passes all the tests. For this, CIGAR uses a so-called ‘initiation prompt’. If the LLM fails to suggest a plausible patch in its first invocation, CIGAR proceeds by using improvement prompts. In the plausible patch multiplication step, after the first plausible patch is generated, CIGAR asks the LLM to produce alternative plausible patches.

Plausible Patch Search Initiation: CIGAR starts with an initiation prompt, with the goal of exploring the search space for a first plausible patch. This prompt consists of three core sections: 1. A one-shot example of fixing a bug, to ensure that CIGAR takes advantage of the in-context learning to understand the expected

format; 2. The buggy function, which is the main part of the input; 3. The test failure details, with valuable extra information about the bug under repair. To present the test failure, the initiation prompt includes the test method name, the assertion that fails, and the assertion error.

The initiation prompt is sent to the model and the LLM is asked to generate 50 samples. Note that at each LLM invocation, CIGAR seeks to generate a diverse set of patches with minimal cost. This is an integral part of efficient search space exploration. Per our preliminary experiments, generating 50 samples per request is the sweet spot that gives us the most diverse set of patches without generating too many repetitive responses which cost output tokens for nothing. After receiving the initial patches in response to the initial LLM invocation, all these patches are tested. If there is any plausible patch among the 50 initial patches, it is directly sent to the plausible patch multiplication step. Otherwise, the first patch without compilation errors together with its test failure message is used for partial patch improvement.

Partial Patch Improvement: The goal of partial patch improvement is to get a plausible patch by improving patches found during the plausible patch search initiation phase. For this purpose, CIGAR takes an iterative approach for generating a plausible patch: until the model generates a plausible patch, CIGAR invokes the LLM with improvement prompts. The improvement prompt gives the LLM a concise feedback on its previous responses. This ensures that the model looks for the plausible patch at different parts of the search space with a low token cost. The partial patches generated in previous iterations are grouped by their failure message. Patches in each of these groups are added next to each other, and their common failure message comes after them. Using this method, we summarize previously generated patches without repeating test failure messages. CIGAR considers as many previous patches as it is possible in a single prompt without going beyond the LLM’s prompt token limit.

Rebooting the Plausible Patch Search: The LLM is reinvoked with improvement prompts as long as no plausible patch is generated, and we have not exceeded *max_invoke* invocations. *max_invoke* is a configurable parameter in CIGAR, set to ten by default per our pilot experiment. As soon as an improvement prompt generates a plausible patch, CIGAR sends it to the plausible patch multiplication step. In case the LLM fails to synthesize a plausible patch after *max_invoke* invocations, CIGAR reboots the whole repair process. This means ignoring all the patches generated before and starting from the initiation prompt. After each reboot, a new *round* of repair starts. Rebooting the repair process enables CIGAR to explore an even larger repair space by starting from a different initial seed, according to the model temperature. For this, CIGAR uses a high temperature, which maximizes exploration. The rebooting strategy with high temperature leads to initiating the repair process from a radically different

hidden state, meaning exploring an undiscovered part of the repair space. As a part of our token minimization approach, this helps CIGAR avoid using too many tokens for exploring a limited and fruitless part of the search space.

Plausible Patch Multiplication: After the first plausible patch is generated with an initiation or an improvement prompt, CIGAR tries to generate new distinct plausible patches. By generating multiple plausible patches, CIGAR has a better chance of producing a correct patch as well. The patch multiplication process also works iteratively. In this step, the prompt contains a summary of the generated plausible patches and a call to action to generate different patches. This prompt includes as many of the recently generated plausible patches as possible without exceeding the LLM’s prompt token limit. This makes the LLM amplify the set of existing patches with patches that are distinct from the previous ones, with a reduced token budget. CIGAR invokes the LLM with multiplication prompts and collects the responses five times. We aim for five multiplication tries to explore a significant part of the plausible patch search space without generating many repetitive patches.

Finally, the full list of all generated plausible patches is reported to the user.

Implementation: CIGAR is implemented in Python and uses OpenAI “gpt-3.5-turbo-0301” as the underlying LLM with sampling temperature of 1. This high temperature adds a notable level of randomness to LLM’s output. This is essential for exploring different parts of the repair space with CIGAR’s reboots and patch multiplication techniques.

Evaluation of CIGAR Effectiveness (EX_1)

Regardless of CIGAR’s token efficiency, the main goal of any APR approach is fixing bugs. Therefore, we first assess CIGAR’s effectiveness. For this purpose, we run CIGAR on a dataset of real-world test-suite based bugs and compare it against three state-of-the-art LLM-based APR approaches: *nl2fix*, STEAM, and CHATREPAIR. We also use CIGAR’s reboot strategy in this experiment. At each round, CIGAR succeeds to generate plausible patches for a group of bugs in our dataset. After each reboot, we only run CIGAR on bugs for which no plausible patch is generated yet. We continue rebooting until CIGAR does not generate a plausible patch for any of the remaining bugs.

Dataset: We use DEFECTS4J [30] and HUMANEVAL-JAVA [21] as the benchmarks of our experiment. DEFECTS4J is the most commonly used benchmark for APR. It enables us to compare against performance metrics available in the published papers or their replication packages. We also consider HUMANEVAL-JAVA [21] to address the potential data leakage concern. HUMANEVAL-JAVA is a benchmark designed fore being more recent than the training data of the

`gpt-3.5-turbo-0301` model. Therefore, a good performance on HUMAN-EVAL-JAVA indicates our results are not susceptible to data leakage. As CIGAR fixes single-function bugs, we only consider bugs of this type. In particular, we consider 162 bugs from HUMAN-EVAL-JAVA and 267 bugs from six DEFECTS4J projects, namely “Chart”, “Closure”, “Lang”, “Math”, “Mockito”, and “Time”. Overall, our experiments consider 429 single-function bugs.

Evaluation Metrics: We use two metrics for this comparison: the number of bugs for which an approach generates a plausible patch, and the number of bugs for which an approach generates a correct patch. We consider a patch to be correct if it has the same AST as the ground-truth bug-fixing patch.

Experimental Results: Table 3.3 shows the results of running CIGAR on our dataset and compares it with the state-of-the-art tools. The table presents the results for DEFECTS4J and HUMAN-EVAL-JAVA bugs both separately and combined. Our aforementioned reboot strategy leads to running CIGAR for 12 rounds. The results for three different rounds is presented in Table 3.3. “#Bugs” represents the total number of bugs on which the APR tool is run. The “#Plausible” and “#Correct (EM)” columns show the number of bugs for which the APR has generated a plausible and a correct patch, respectively.

CIGAR is able to generate a plausible patch for 69.2% of DEFECTS4J bugs and 93.8% of HUMAN-EVAL-JAVA bugs, which is arguably high. In total, CIGAR produces plausible patches for 78.5% (337/429) of the bugs in our dataset. This is higher than the best performance by state-of-the-art, CHATREPAIR, which generates plausible patches for 59.9% (160/267) of DEFECTS4J bugs and 93.2% (151/162) of HUMAN-EVAL-JAVA. In the first round R1, CIGAR generates a plausible patch for 66.8% (287/429) of the bugs, and this number increases at each round until the twelfth round. This shows the effectiveness of the reboot strategy adopted by CIGAR, which maximizes exploration using a new random seed thanks to the temperature mechanism.











Generating a correct patch is the ultimate goal of any APR tool. CIGAR outperforms all other tools by generating a correct patch for 25.8% (69/267) of DEFECTS4J bugs and 93.8% (152/162) of HUMAN-EVAL-JAVA bugs. This proves the overall end-to-end effectiveness of CIGAR. The number of bugs with a correct patch also increases over rounds, again demonstrating the importance of rebooting. CIGAR surpasses all existing tools on both DEFECTS4J and HUMAN-EVAL-JAVA in terms of correct patches at round three (R3).

Overall, this experiment demonstrate that CIGAR’s novel iterative prompting pipeline effectively explore the search space of LLMs to find correct patches.

Table 3.3: Effectiveness of CiGAR vs recent LLM-based program repair tools on DEFECTS4J.

Tool	DEFECTS4J			HUMANEval-JAVA			ALL		
	#Bugs	#Plausible	#Correct (EM)	#Bugs	#Plausible	#Correct (EM)	#Bugs	#Plausible	#Correct (EM)
<i>nJFix</i>	283	53.3% (151/283)	11.3% (32/283)	162	93.2% (151/162)	52.4% (85/162)	429	72.4% (311/429)	32.1% (138/429)
STEAM	260	—	25.0% (65/260)						
CHATREPAIR	267	59.9% (160/267)	19.8% (53/267)						
CiGAR_R1	267	56.5% (151/267)	23.9% (64/267)	162	83.9% (136/162)	61.1% (99/162)	429	66.8% (287/429)	33.7% (145/429)
CiGAR_R3	267	61.4% (164/267)	25.4% (68/267)	162	90.1% (146/162)	62.9% (102/162)	429	72.2% (310/429)	39.6% (170/429)
CiGAR_R12	267	69.2% (185/267)	25.8% (69/267)	162	93.8% (152/162)	62.9% (102/162)	429	78.5% (337/429)	39.8% (171/429)

Table 3.4: Token cost of CIGAR and CHATREPAIR on various types of bugs. Lower cost indicates better token cost efficiency.

Bugs_Fixed_By	CHC	CIC	CHC_AVG	CIC_AVG	Saving
CHATREPAIR (13)	8.6M	1.5M	 661K	 115K	–
CIGAR (46)	24.4M	1.5M	 530K	 32K	–
Both (125)	76M	2.6M	 608K	 20K	96%
Neither (245)	95.3M	49.1M	 388K	 200K	48%
Total (429)	204.3M	54.9M	 467K	 127K	73%

Evaluation of CIGAR Efficiency (EX_2): We conduct a second experiment to assess if the strategies adopted by CIGAR make it an efficient APR approach. In this regard, we need to compare CIGAR’s token cost against our baselines on the same dataset as the one used in EX_1. As the three baselines, *nl2fix*, STEAM, and CHATREPAIR are not open-source, we have to reimplement them to be able to accurately measure their token cost. CHATREPAIR is more effective than *nl2fix* and more token efficient than STEAM. Therefore, we decide to reimplement and consider CHATREPAIR for this experiment.

We compare CIGAR’s total token cost against the total token cost of CHATREPAIR. We consider the token cost to be the sum of the number of tokens sent to the LLM in the prompt or received from it in its responses. We also study the token cost of CIGAR and CHATREPAIR on bugs that are fixed vs not-fixed by each tools per our RQ1 experimental results. This sheds light on a different aspect of the efficiency of each tool, as it splits the token cost to fruitful ones leading to a correct patch and unfruitful ones, where the token price to pay is a pure loss.

Experimental Results: We compute the token cost of each approach on bugs of different types. Table 3.4 shows the result of this experiment. The “Bugs_Fixed_By” column indicates the type of bugs according to the tool(s) that fix them. The number in brackets in this column is the number of bugs of the respective type. “CHC” and “CIC” show the token cost of CHATREPAIR and CIGAR on each type of bug, respectively. Also, “CHC_AVG” and “CIC_AVG” represent CHATREPAIR’s and CIGAR’s average token cost per bug for each type. Finally, the “Saving” column indicates the percentage of saved token cost by CIGAR compared to CHATREPAIR.

In total, CHATREPAIR spends 467K tokens on average, while CIGAR spends 127K. This means CIGAR improves the token cost by 73% (149.4M/204.3M), while, per EX_1, it outperforms CHATREPAIR in terms of the number of plausible/correct generated patches, demonstrating a win-win situation. This is explained by of our token minimization strategies: asking for as many samples as possible at each LLM invocation, timely reboots, and summarizing the previous patches in the prompt. To sum up, CIGAR improves on both aspects: 1) it

enables us to have more distinct patches and 2) for fewer tokens.

In Table 3.4, we see that CIGAR saves 96% (73.4M/76M) of token cost on bugs that are fixed by both. We also see that CIGAR spends 32K tokens on average for the 46 bugs that are only fixed by CIGAR, while CHATREPAIR spends 661K tokens on average for the 13 bugs only fixed by CHATREPAIR. All this data shows, when the tool is able to find a correct patch, CIGAR works much more efficiently than CHATREPAIR. By manually checking our experimental data, we see the reason is that CHATREPAIR uses up to 199 LLM invocations after generating the first plausible patch to produce alternative plausible patches one by one. In contrast, CIGAR only uses five LLM invocations for plausible patch multiplication and generates up to 50 patches at each invocation. We see that CIGAR’s summarization of previous patches in each invocation and using a high temperature is effective: it leads the LLM to synthesize novel patches without using too many invocations and causing unnecessary cost.

Finally, we consider the 245 bugs that are fixed by neither of the tools. CHATREPAIR’s token cost on these bugs is 95.3 million and CIGAR’s token cost is 49.1 million. On average, CHATREPAIR spends 388K (95.3M/245) tokens on each of these bugs and CIGAR spends 200K (49.1/245). This means CIGAR saves 48% (188K/388K) of the token cost for unfixable bugs. The token cost difference on unfixable bugs is because CIGAR summarizes previously generated partial patches, while CHATREPAIR includes the whole previously generated patches in its new invocations.

Overall, this experiment shows that CIGAR is particularly cost-efficient when it is successful at generating a correct patch for a bug, with a token total price of only 4% on average compared to the baseline, representing a saving of 96%.

3.1.3 Conclusion

Based on our experiments explained in subsection 3.1.1 and subsection 3.1.2, our contributions lead to a state-of-the-art APR approach in terms of both effectiveness and efficiency: CIGAR. CIGAR takes advantage of LLMs understanding of bug context to create an effective APR tool. At the same time, it also employs an advanced three-stage prompting strategy and delicately designed reboot policy to minimize the token cost of using LLMs. In sum, our proposed methods make APR patch generation significantly more efficient.

3.2 P2: Synthesis and Usage of Differencing Data for Effective Automated Patch Assessment

As described in section 2.3, most automated patch assessment methods heavily depend on the availability of data that distinguish between program versions. This includes techniques that compare the buggy version and patched versions to detect regression issues [153], or compare the behavior of patches to identify the

ones more likely to be correct [15]. Most recently, Martinez et al. [9] employ generated tests to separate patches and cluster them into different groups. Previous works utilize old techniques for differentiating between program versions, such as generating tests with EvoSuite [19] and Randoop [156].

We make two contributions to help making automated patch assessment effective. The first contribution is a lightweight tool, called LIGHTER [33], an efficient tool that uses historical data to estimate the potential of different fix templates. LIGHTER compares a large history of human-made commits against template-based APR strategies to estimate the potential of their search spaces. The second contribution is MOKAV [204], an LLM-based execution-driven approach for generating difference exposing test inputs. We discuss these two contributions in the following.

3.2.1 Contribution 3: Estimating the Potential of Program Repair Search Spaces with Commit Analysis

Researchers have proposed techniques to mine the history of real-world projects for suggesting effective APR fix templates [79]. As explained in section 2.2, these techniques make the patch generation of template-based APR approaches precise as they contain patches similar to human-made bug-fixing commits. The idea behind mining commit histories is as follows: a template-based APR approach that mimics many human-written past commits has a strong potential to generate useful patches. Based on this observation, we build LIGHTER. LIGHTER employs commit analysis for automated assessment of template-based APR search spaces.

LIGHTER is a lightweight tool that estimates the potential of APR fix templates with a fully static analysis. Instead of executing a repair system to check if its templates actually fix certain bugs, it analyzes whether real world bug-fixing commits lie in the their repair search space. For this purpose, LIGHTER encodes APR search spaces by specifying the repair strategies they employ. Next, it uses the specifications to check whether past commits lie in repair search spaces. For a repair approach, including many human-written past commits in its search space indicates its potential to generate useful patches. In particular, LIGHTER specifies the search spaces of 8 notable template-based APR approaches: Arja [23], Cardumen [13], Elixir [24], GenProg [25], jMutRepair [26], Kali [27], Nopol [14], and NPEfix [28]. LIGHTER also provides a framework to readily specify and assess the search space of future template-based APR approaches.

Terminology

LIGHTER computes the number of human-written patches that lie in the search space of template-based APR approaches. For this, it computes their *commit coverage*. Here we present the terminology that we use in this work leading to the definition of *commit coverage*.

Repair Operator: A type of atomic change that is applied on the buggy program to repair the bug. For example, removing a statement from the source code is an operator used by Kali [27].

Repair Strategy: A set of repair operators applied in conjunction by a repair approach to the buggy version of a program. For example, one of the strategies employed by NPEfix [28] is “skip method” (e.g., see Listing 2.1). Per this strategy, an if-statement is added before a suspicious statement. The corresponding if-condition checks whether a variable used by the suspicious statement is equal to null. If the if-condition holds, a return statement is executed.

Repair Ingredient: An existing source code fragment that is reused by a repair approach to fix the bug [102, 205]. For example, in one of its repair strategies, GenProg [25] creates a candidate patch by replacing a suspicious statement by another existing statement written elsewhere in the program. The latter is the *ingredient* of the candidate patch. Note that ingredients can have different granularities. For example, in GenProg, an ingredient is a statement, in NPEfix [28] it is a variable, and in Cardumen [13] it is an expression.

Scope of Ingredients: The scope of ingredients is the parts of program that are considered for extracting repair ingredients [102, 205]. For example, jGenProg [26] can replace an old statement s (written in file f from package p) with a new one, according to three different scopes: 1) same file (i.e., f), 2) same package (i.e., from any file belonging to p), and 3) same program.

Search Space of Repair Approach: Let us assume a repair approach r with certain repair strategies and a scope of ingredients. When a program is given as the input, the search space of r is the set of all patches that r can generate given the strategies and scope of ingredients [46].

Repair-space Commit: Given a commit c that transforms the *old_version* of a program into its *new_version* and a repair approach r , we say that commit $c = (old_version, new_version)$ is a *repair-space-commit* for r if and only if *new_version* is in the search space of r when *old_version* is given as the input.

Again, consider Listing 2.1 as an example. The commit contains a typical NPEfix null check characteristic of its search space. Hence, we say that this commit is a repair-space commit for NPEfix. In this example, line 1 and line 5 are from the *old_version*, while in the *new_version* lines 2-4 are added. According to the definition of NPEfix, *new_version* is in its search space because NPEfix has a “skip method” strategy that is able to produce this patch.

Commit Coverage (CC): The *commit coverage* of repair approach r over a set of commits S is the number of commits in S that are repair-space commit for r divided by the total number of commits in S .

We consider real human-written patches to be useful patches. Therefore, if a repair approach has a high commit coverage over a large dataset of human-written patches, this indicates the potential usefulness of the fix templates of the repair approach.

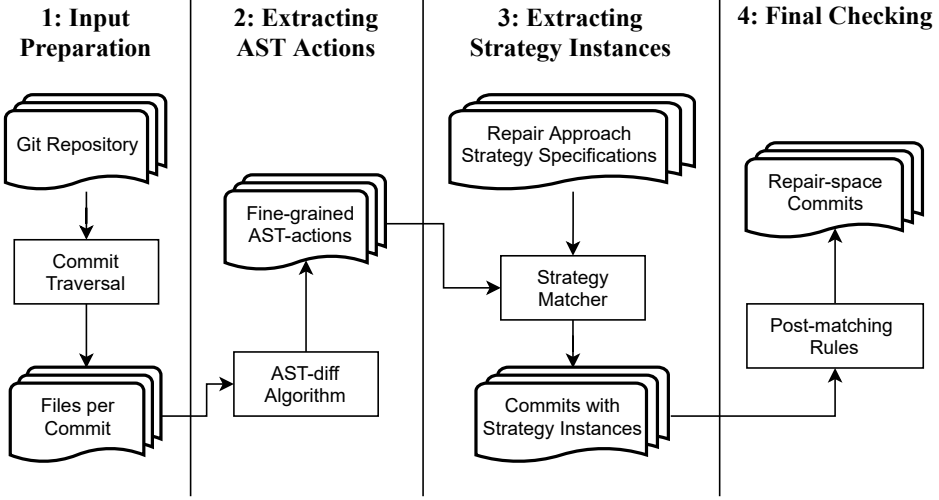


Figure 3.4: Overview of LIGHTER’s approach.

LIGHTER’s Approach

LIGHTER is a framework to detect repair-space commits. When the repair-space commits are detected, computing the commit coverages is trivial. In this framework, we encode each repair strategy by specifying *a)* the repair operators from the strategy, expressed using fine-grained code changes, and *b)* the rules that those changes must respect (e.g., the code introduced by a change is a valid ingredient according to a given scope).

Figure 3.4 shows an overview of how the proposed approach works. The whole process consists of four steps. *1)* Input preparation: for a given Git repository, we identify the updated files for each commit. *2)* Extracting AST actions: for each updated file, the actions that transform the AST of the old version into the new one are extracted. *3)* Extracting strategy instances: the updated files whose corresponding AST actions match a “strategy specification” are determined. We design the strategy specifications to model the repair strategies employed by the considered repair approaches. *4)* Final check: the commits whose updated files match strategy specifications are checked for additional constraints. The result of this last step are the detected repair-space commits.

Extracting AST Actions: For a given commit c , LIGHTER checks that it only updates one file. Changes in multiple updated files cannot be covered by a single strategy instance, while we only target single-instance fixes that lie in the repair search space. If c has one updated file, LIGHTER gets that file f and constructs a pair of files $\langle f_p, f_n \rangle$, where $\langle f_p \rangle$ is the version of f previous to c , and $\langle f_n \rangle$ is the new version obtained after c . Next, LIGHTER uses GumTree [189] to extract

```

1 <pattern name="binary_upd">
2 <entity id="1" type="If"/>
3 <entity id="2" type="BinaryOperator">
4 <parent parentId="1">
5 </entity>
6 <action entityId="2" type="UPD"/>
7 </pattern>

```

Listing 3.2: One of the strategy specifications for jMutRepair.

the AST modifications between f_p and f_n . GumTree outputs an *edit script* (ES), a list of actions that transforms the AST f_p into f_n . In GumTree, there are four types of action on AST: 1) update, which changes the value of an AST node, 2) insert, which inserts a new AST node, 3) delete, which deletes an existing AST node, and 4) move, which moves an AST node and makes it child of another node. The extracted ES is passed to extracting strategy instances step.

Extracting Strategy Instances: For a repair approach, we come up with one or more strategy specifications that define its search space. Strategy specifications are abstract representations of the repair strategies employed by repair approaches. If AST actions in an edit script ES match with a strategy specification s , we say that the ES is an *instance* of the s .

The specifications are represented in the *change pattern specification language* of [206]. A strategy specification consists of a set of *actions*, and each action is performed on an *entity*. The types of actions of specifications are the same as the types of AST actions that GumTree extracts (*update*, *insert*, *delete*, and *move*). Finally, a strategy specification can also define parenthood relations between entities.

For example, Listing 3.2 is a specification that corresponds to a repair strategy used by jMutRepair [26]. According to this strategy, a binary operator inside an if-condition can be changed to another operator. Line 6 of the Listing 3.2 represents the update action. As it is stated, the “entityId” of the subject entity is “2”. Therefore, this action is performed on the entity defined in line 3. As shown in the specification, the type of this entity is “BinaryOperator” and the id of its parent is “1” (see line 4). Finally, the parent entity is defined in line 2 and as it is mentioned there its type is “If”.

For each strategy specification s of a repair approach, LIGHTER checks if s matches with the AST actions (ES) previously computed. To this end, for each action A_p specified in s , we check whether there exists an actual action in ES that affects the nodes specified by A_p . The details of the matching process can be found in the study of Martinez et al. [206]. Note that LIGHTER considers a commit to be in the search space of repair approach r , only if all the changes in the commit are covered by a single strategy instance of r .

Final Checking: In order to make sure that the source code changes from the identified commits lie in the search space of detected repair approaches, LIGHTER also checks particular rules that repair approaches follow for generating patches. We call these rules the *post-matching rules*. These rules determine how a repair approach synthesizes new code. As an example, Cardumen [13] considers all the variables and literals in the scope as repair ingredients. Next, it takes an existing *expression* and replaces its variables and literals with extracted ingredients of the same type to make a new expression. This new expression is then used to generate a patch. The commits that follow the post-matching rules are considered as the detected repair-space commits.

Assessing the Search Space of Considered APR Approaches (EX_1)

We use LIGHTER to assess the potential of the search space of our eight considered APR approaches. For this, we compute the commit coverage of these approaches on a large dataset of real-world commits. We also conduct a manual analysis to decide if the repair-space commits of each of these approaches are bug-fixing commits or not. This analysis clarifies whether the search space of an approach actually corresponds to human-made fixes. To assess the performance of LIGHTER, we also compute the average time it spends to check if a commit is in the search space of a given APR approach.

Dataset: For this experiment, we create a dataset of recent commits from 72 repositories considered in the BEARS [34] benchmark. We consider BEARS repositories, as BEARS is the dataset of Java bugs with largest number of repositories. From each repository, we consider the 1,000 last commits. Among the 72 repositories in BEARS, 43 have more than 1,000 commits and 29 have fewer commits. In total, we collect 55,309 commits. We further filter these commits by only keeping commits that change exactly one Java source file. The result is a dataset of 7,583 commits, which we call PB.

Experimental Results: Table 3.5 shows the results of this experiment. The “#RSC” column shows the number commits that LIGHTER detects as repair-space commits. “%CC” presents the commit coverage, which is equal to the percentage of 7,583 human-written commits that are considered to be repair-space commits. “BF”, “NBF”, and “UD” indicate the and percentage of repair-space commits labeled as “bug-fix”, “not-bug-fix”, and “undecided”, respectively. Finally, the “Exec. time” column represents how many seconds it takes on average for LIGHTER to check if a commit is in the search space of the corresponding approach. For example, 263 of commits are detected to be in the search space of Arja, which means Arja covers 3.46% of the commits. Moreover, 51% of Arja’s repair-space commits are labeled as “bug-fix” commits by the annotators.

In total, 747/7,583 (9.85%) commits are detected as repair-space commits for, at least, one of the repair approaches. Among the considered repair approaches,

Table 3.5: The presence of repair-space commits in 72 open-source projects.

Approach	#RSC ^a	%CC ^c	BF ^d	NBF ^d	UD ^d	Exec. Time
Arja	263	3.46	51%	38%	11%	0.76s
Cardumen	219	2.88	63%	29%	8%	0.33s
Elixir	369	4.86	67%	24%	9%	1.72s
GenProg	181	2.38	46%	42%	12%	0.77s
jMutRepair	7	0.09	86%	14%	0%	0.87s
Kali	117	1.54	31%	56%	13%	0.29s
Nopol	174	2.29	81%	13%	6%	1.34s
NPEfix	33	0.43	90%	3%	7%	0.46s
All	747 ^b	9.85	62%	29%	9%	0.81

^a RSC stands for “repair-space commits”. This column shows how many of the 55,309 commits that are analyzed against the search space of all tools are detected as repair-space commits of this approach. ^b 747 commits are detected repair-space commits for at least one repair approach. Note that this is not the sum of numbers in this column.

^c %CC is the commit coverage of the corresponding approach, it shows the percentage of commits with changes in exactly one source file that lie in the search space of corresponding repair approach. The total number of these commits is 7,583. For example, Elixir covers 4.86% (369/7,583) of commits with changes in exactly one source file. ^d BF, NBF, and DN represent the number of repair-space commits labelled as “bug-fix”, “not-bug-fix”, and “UD”, respectively.

the top two approaches in terms of the commit coverage are Arja and Elixir. Given the strategies used by these approaches, this confirms the results of Martinez et al. [205] showing that in a significant number of commits all the new lines are copied from the previous versions of the same file.

The majority (62%) of the detected repair-space commits are labeled as “bug-fix”. This is in line with the fact that the encoded repair strategies are indeed related to the activity of bug fixing. Interestingly, there is also a notable portion of the repair-space commits (29%) that are not considered as bug-fixing, yet can likely be generated by a repair approach. This suggests that the considered repair approaches can also be used for purposes other than bug-fixing. We manually analyze them and identify that there are two common types of “not-bug-fix” repair-space commits: 1) commits that only change logging outputs, and 2) commits that remove unused code. This indicates that the fix templates employed by APR approaches can potentially be used for tasks other than program repair as well. There are also 9% of repair-space commits that our analysts could not determine if they were bug-fixing or not.

The last column of Table 3.5 shows that the average time spent to check if a commit is in the search space of a repair approach is 0.81 second. This indicates that LIGHTER can scale to large repositories. For instance, analyzing

Table 3.6: Recall and precision of LIGHTER per our manual analysis.

Approach	Precision	Recall
Arja	0.96	0.90
Cardumen	0.83	0.91
Elixir	0.70	0.83
GenProg	0.80	0.87
jMutRepair	0.71	1
Kali	0.96	0.88
Nopol	0.60	0.98
NPEfix	0.60	0.97
Total	0.77	0.92

3.3k commits (the maximum number of commits for 99% of Java projects on Github) against the search space of all tools would take approximately 6 hours, which is acceptable given that it is one-shot computation task.

Assessing LIGHTER’s Accuracy (EX_2)

We conduct a manual experiment to measure LIGHTER’s precision and recall. To measure the precision of LIGHTER, we randomly select 30 repair-space commits for each tool in the PB dataset. Next, we carry out a manual analysis to decide if the detected repair-space commits actually lie in the search space of corresponding repair approaches or not. Each repair-space commits is annotated by two APR experts. If the first two annotations conflict with each other, a third analyst annotates to break the tie.

In our manual analysis for recall measurement, we assess how many of the patches actually generated by a repair approach are correctly detected by LIGHTER. For this, we consider a *ground-truth* benchmark of 729 patches generated by our eight considered APR approaches for Defects4J bugs. This benchmark is collected from two previous studies, DRR [15] and NPEfix [28]. We consider the recall of LIGHTER as the percentage of the patches in the *ground-truth* benchmark that LIGHTER detects as repair-space commits.

Since LIGHTER is the first tool of its type, there is no other work that we can directly compare against. However, a close tool is PPD (Patch Pattern Detector), which detects instances of repair patterns [207]. The repair patterns that PPD looks for are extracted from the code changes in Defects4J dataset and the tool is also evaluated on Defects4J. We compare the recall of our tool against PPD.

Experimental Results: Table 3.6 shows the result of our manual analysis of LIGHTER’s precision and recall per APR approach. Thanks to the careful design of the matching criteria, the precision of LIGHTER is 0.77. It is never lower than

0.60 for any of the considered repair approaches. Also, on the 729 ground-truth patches, we compute that the recall of LIGHTER is 0.92, which is on par or higher than the closest related tools [207]. Per repair approach, the recall has a minimum 0.83 and a median of 0.90, which is consistently high. Therefore, we conclude that LIGHTER can be trusted in terms of detecting commits that actually lie in the search space of program repair approaches.

Implication for Researchers and Practitioners

Our two experiments show that LIGHTER is a lightweight accurate tool for conducting a fast evaluation of the breadth of the search space of different fix templates. We conclude that LIGHTER can be used by program repair researchers to assess the search space they envision without taking the hard path of fully implementing a tool. More specifically, LIGHTER enables researchers to measure the extent to which human-written commits are covered by their envisioned fix templates without full execution. LIGHTER also allows practitioners to quickly measure how many of their historical changes lie in the search space of specific APR approaches. This can be done without requiring heavy changes in the target project, configuring of a repair tool or having failing test cases for all past commits. This gives developers an estimation of their own bug-fixes that are in the search space of certain repair tools.

3.2.2 Contribution 4: Execution-driven Differential Testing with LLMs

Concrete data that exposes differences between various program versions plays an integral role in automated patch assessment. To obtain such data, researchers have used traditional test generation tools, such as EvoSuite [19] and Randoop [156]. They use these tools to produce tests that reveal behavioral differences between program versions [9, 153]. There are two limitations with these approaches. First, the main goal of test generation tools adopted in these approaches are not aimed at generating difference exposing tests. For example, EvoSuite employs a genetic algorithm to generate tests that maximize branch coverage. Second, existing studies do not take advantage of most advanced test generation techniques that utilize LLMs [208].

In this contribution, we propose MOKAV, a novel execution-driven tool that leverages LLMs to generate difference exposing tests (DETs). Given two program versions P & Q, a DET is a test input that results in different outputs on P & Q. Therefore, a DET exposes the functional difference between two versions. MOKAV iteratively requests an LLM to generate a DET based on two programs P & Q. At each iteration, MOKAV provides three pieces of information in its prompt to the LLM. First, an example test input that produces the same output on P & Q. This example test hints at the model regarding the type and structure of inputs acceptable by P & Q. Second, MOKAV runs the example test on P & Q

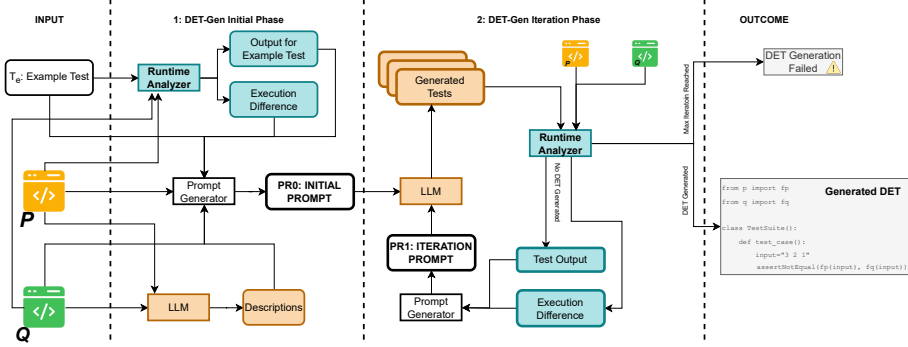


Figure 3.5: Overview of MOKAV. MOKAV exposes functional differences between two programs P and Q by feeding LLMs with execution feedback.

and adds their identical output to the prompt. The output produced by P & Q on the example test provides extra information about their functionality. Third, MOKAV collects variable values while running the example test on P & Q and adds detected disparities in the prompt. Using this fine-grained execution data, MOKAV steers the LLM towards parts of the input space that contain DETs. Using its execution-driven and iterative method, MOKAV generates DETs that can help automated patch assessment approaches.

Terminology

As discussed in subsection 3.2.2, MOKAV generates DETs, tests that expose functional differences. Here we clarify our definition of DETs and distinguish between functional and execution difference as follows.

Difference Exposing Test (DET): Let $P: I \rightarrow O$ and $Q: I \rightarrow O$ be two programs operating deterministically on the same input/output space, where I is the input domain of P and Q and O is the output domain. Then, *det* is a difference exposing test iff $det \in I \wedge P(det) \neq Q(det)$.

Note that per our definition, a DET detects *functional differences* between P and Q, meaning they produce different outputs for a given input. In this work, we consider functional difference to be distinct from the *execution difference*. P and Q have an execution difference if, for a given input, there is a variable v used in both P and Q where the set of values v takes on in P is different from the set of values v takes on in Q. In essence, functional difference relates to external differences between P and Q that can be identified by an external observer; in contrast, execution difference pertains to internal differences between P and Q that are detected solely through monitoring the internal states of P and Q. The goal of this contribution is to design a novel, effective approach for DET generation.

```

1 Diff between P & Q:
2 x = input().split ()
3 x.sort ()
4 for i in range(len(x)):
5     x[i] = int(x[i])
6 x.sort ()
7 n = 100
8 for j in range(2):
9     if ((x[j] + x[(j + 1)]) > x[(j + 2)]):
10         n = 300
11     elif ((x[j] + x[(j + 1)]) == x[(j + 2)]):
12         n = max(n, 200)
13     else:
14         n = max(n, 100)
15 if (n == 300):
16     print ('TRIANGLE')
17 elif (n == 200):
18     print ('SIGMENT')
19     print ('SEGMENT')
20 else:
21     print ('IMPOSSIBLE')
22 Example test:
23 Input:
24     "4_2_1_3"
25 Output:
26     P: "TRIANGLE"
27     Q: "TRIANGLE"
28
29 Generated Difference-exposing test (DET):
30 Input:
31     "5_2_1_3"
32 Output:
33     P: "SIGMENT"
34     Q: "SEGMENT"

```

Listing 3.3: Two versions of a program that takes four numbers, sorts them, and checks if the first three or last three form a tirangle. An example test with identical output on P & Q and a DET generated by MOKAV are also shown.

MOKAV's Approach

MOKAV takes an iterative approach to DET generation. It asks the LLM to generate a set of test inputs, gives feedback to the LLM regarding the execution data captured in the generated test inputs, and loops until a DET is generated or a maximum budget is reached.

Figure 3.5 overviews how MOKAV works. MOKAV takes as input two versions of a program (P and Q) as well as a sample test input, for which both versions produce the same output. We call this input an *example test*. In the initial phase, MOKAV creates an initial prompt based on P and Q, the example test, the execution data extracted while running the example test, and an LLM-generated

description of the intention of P and Q. The description of P and Q clarifies the program intent in natural language, which is the language more familiar for many LLMs. Next, in the iteration phase, MOKAV uses an LLM to generate new test inputs and iteratively gives execution based feedback to the LLM until it generates a DET. The generated DET is reported as the outcome of MOKAV.

Listing 3.3 shows an example pair of programs given to MOKAV and the DET it generates. In this example, P and Q are two real-world programs by a single user for a problem in the CodeForces² code competition. A correct program should sort the input numbers and check if the first three or the last three make up a triangle, a segment, or neither. The example test is “4 2 1 3”, for which both versions output “TRIANGLE”. Given P, Q, and the example test, MOKAV generates a new input data: “5 2 1 3”. On this input, P’s output is “SIGMENT” with a misspelling, while Q correctly outputs “SEGMENT”. This means MOKAV successfully generates a DET.

Input: The input to MOKAV is a tuple in the form of (P, Q, T_e) . P and Q are a pair of versions of the program for which MOKAV should generate a DET. The last input to MOKAV, T_e , is an example test with the same corresponding output on both P and Q, which means it does not expose functional differences. This example test is used to capture execution differences between P and Q and also to show the type of the desired test input to the LLM.

Initial Phase: In the initial phase, MOKAV invokes the LLM with an *initial prompt*. The initial prompt consists of (1) the P and Q versions of the program, (2) the description of P and Q, generated by the LLM, (3) the example test and its corresponding output, (4) the *execution difference* detected while running the example test, and (5) text to specify the format of response that is expected. Among these elements, the output of example test and execution difference require further computation as follows.

MOKAV executes the example test on P and Q to obtain the corresponding output and identify execution differences. Note that the output for the example test must be the same on both versions. To identify the execution differences the MOKAV takes three steps as follows. First, it finds the variables that are common between P and Q. Next, it uses spotflow³ to collect variable values during the execution of P and Q. Finally, MOKAV identifies the first time a common variable gets a value in one of the versions, while that variable value never occurs in the other version. Such variable values are called *unique variable values* [209], and the MOKAV considers them as the detected execution difference. The output for an example test and the detected execution differences clarify the behavior of the programs, which is crucial for DET generation.

²<https://codeforces.com/problemset/problem/6/A>

³<https://github.com/andrehora/spotflow>

Given the initial prompt, the LLM generates a first candidate DET. MOKAV executes both P and Q on the generated input to verify the presence of a functional difference. If that happens, MOKAV generates a DET with the initial prompt directly. Otherwise, it goes to the iteration phase.

Iteration Phase: MOKAV iteratively prompts the LLM until either it generates a DET or the maximum number of iterations is reached. At each iteration, after the model replies with a set of generated tests, the tests are executed on P and Q. If a test produces different outputs on P and Q, it is a valid DET, MOKAV has succeeded, and we exit. Otherwise, MOKAV tries again by requesting the LLM with an iteration prompt. To create iteration prompt, MOKAV selects the first generated test and uses produces execution based feedback for the selected test. This feedback contains the output of P and Q on the selected test and the execution difference detected during the execution of this test. MOKAV gives feedback only on the first generated test to avoid exceeding the prompt size limit. At the end of the iteration phase, the result is either a generated test that exposes the functional difference or a failure message that the maximum number of iterations is reached.

Outcome: The generated difference exposing test is presented to the user in the form of a unit test class. This test case imports P and Q as two functions `fp` and `fq`. Then a test case method is passes the generated DET to `fp` and `fq` and asserts that their outputs are not equal (for example, see the outcome section of Figure 3.5).

Implementation: MOKAV is implemented in Python, and can be configured with different LLMs. By default, it supports OpenAI’s GPT models and `CodeLLama-Instruct`. MOKAV uses three important values that can be configured for each execution: the maximum number of iterations, the number of samples requested from the LLM at each invocation, and the temperature of the LLM. The default configuration is 10 iterations, 10 samples, and temperature=1.

Evaluation of MOKAV’s DET Generation Effectiveness

We assess the effectiveness of MOKAV by computing for what percentage of pair of programs with guaranteed functional difference it generates a DET. We compare MOKAV’s effectiveness with PYNGUIN [210] and Differential Prompting (DPP) [211]. PYNGUIN is a traditional test generation tool that employs genetic algorithms to generate tests with maximum coverage over a Python program. DPP is an LLM-based tool that prompts the LLM for test generation in a single iteration. DPP takes a single buggy Python program and generates fault-inducing tests that expose the bug. By comparing MOKAV against DPP and PYNGUIN, we evaluate if our iterative and execution-driven tool that is

Table 3.7: Effectiveness of MOKAV compared with related work. #Success_pair is the number of pairs for which a DET is generated, #TT is the number of all generated tests

Tool	QuixBugs		4det	
	#Success_Pair	#TT	#Success_Pair	#TT
PYNGUIN	50.0% (16/32)	1,176	4.9% (76/1,535)	27,487
DPP	50.0% (16/32)	3,200	37.3% (573/1,535)	153,500
MOKAV (iter 1)	93.6% (30/32)	320	58.9% (905/1,535)	15,350
MOKAV (iter 4)	100.0% (32/32)	340	76.4% (1,173/1,535)	30,260
MOKAV (iter 7)	100.0% (32/32)	340	80.3% (1,234/1,535)	40,240
MOKAV	100.0% (32/32)	340	81.7% (1,255/1,535)	48,930

particularly aimed at DET generation outperforms state-of-the-art Python test generation tools. We collect MOKAV’s effectiveness data at first, third, seventh, and tenth iterations. This data describes if MOKAV’s effectiveness improves with more iterations. In addition to effectiveness, we also compute the total number of tests it generates to measure MOKAV’s cost efficiency.

Dataset: We use two datasets for this experiment. The first one is QuixBugs [29], a dataset of 32 pairs of programs in Python that implement common programming problems. In each pair, one version is buggy and the other one is fixed, which guarantees that the two versions are functionally different. The second dataset is C4DET, a dataset of 1,535 pair of Python programs collected from submissions to Codeforces competitions. In each pair $\langle P, Q \rangle$, P & Q are submissions for the same problem, where P gives wrong answer on a test and Q passes all the tests. This guarantees that there is functional difference between P & Q. We evaluate the effectiveness of MOKAV, DPP, and PYNGUIN on these two datasets of program pairs with guaranteed functional differences.

Experimental Results: Table 3.7 compares the effectiveness of MOKAV with two baselines (PYNGUIN and DPP) on QuixBugs and C4DET. The table presents MOKAV’s effectiveness with 1, 4, and 7 iterations, and its default setting with 10 iterations on the last row. For each dataset, we measure two metrics: the number of program pairs with DETs (“#Success_Pair”) and the total number of generated tests (“#TT”).

According to Table 3.7, MOKAV generates a DET for 58.9% (905/1,535) of C4DET pairs at its first iteration and for 81.7% (1,255/1,535) of pairs at its last iteration, which is the highest performance reported. First, this shows that the iterative approach significantly improves the effectiveness of MOKAV. Second, this result shows that MOKAV outperforms both strong baselines PYNGUIN and DPP, which generate DETs for 4.9% and 37.3% of program pairs, respectively. Our manual investigation reveals that MOKAV outperforms PYNGUIN and DPP mainly because it better understands the type of arguments the programs take.

```

1 The diff between P & Q:
2 s = input()
3 s1 = [i for i in range(len(s)) if ((s[i]=='A') or (s[i]=='E') or (s[i]=='I') or (s[i]
   ]=='O') or (s[i]=='U') or (s[i]=='Y'))]
4 d = 0
5 for i in range((len(s1) - 1)):
6     if ((s1[(i + 1)] - s1[i]) > d):
7         d = (s1[(i + 1)] - s1[i])
8     if ((d == 0) and (len(s1) != 0)):
9         print(max((s1[0] + 1), (len(s) - s1[(- 1)])))
10    elif (len(s1) == 0):
11        print((len(s) + 1))
12    else :
13        d = max(d, (len(s) - s1[(- 1)]))
14        d = max(d, (len(s) - s1[(- 1)]), (s1[0] + 1))
15    print(d)
16
17 DET generated by Mokav:
18 Input:
19 "BYAIAUOIEOAA"
20 Output:
21 P: 1
22 Q: 2

```

Listing 3.4: A pair of programs for which only MOKAV is able to generates a complex DET, after several iterations.

This confirms the importance of using an example test in the prompt to hint the LLM regarding the type and structure of inputs.

We notice that MOKAV, due to its iterative and execution-driven feedback, better understands the semantic differences between P and Q. Listing 3.4 shows an example where MOKAV generates a DET at its final iteration, while PYNGUIN, and DPP fail to do so. P and Q are supposed to compute the distance between certain characters in a string and the beginning and end of a given string, and output the longest distance. P wrongly ignores the distance between the beginning of the string and the first appearance of the considered characters (line 13). MOKAV successfully generates “BYAIAUOIEOAA” as a DET for this pair. The DET is only generated at the last iteration of MOKAV. This indicates that the iterative and execution-driven approach of MOKAV enables it to better understand P and Q’s semantics compared to DPP and PYNGUIN.

The next column (“#TT”) shows the total number of tests generated by each tool. DPP generates 153,500 tests for 1,535 pairs in C4DET, because it asks for a fixed number of 100 tests for each pair. MOKAV generates batches of 10 tests at each iteration only if the previous iterations fail to yield a DET. Consequently, MOKAV generates only 15,350 tests in its first iteration and this number gradually increases to 48,930 at its 10th iteration. This shows that MOKAV’s iterative approach also contributes to its cost efficiency by generating new tests step-by-

step and only when needed.

All the major takeaways from the results on C4DET also hold for QuixBugs. Most importantly, MOKAV outperforms both PYNGUIN and DPP even at its first iteration. MOKAV generates a DET for 100% (32/32) of the program pairs in QuixBugs. This demonstrates the perfect effectiveness of MOKAV on this widely-used dataset.

Implication

Using an iterative execution-driven approach, MOKAV offers an effective technique for generating test inputs that expose functional differences. Notably, MOKAV generates more DETs than the baselines, while generating fewer tests in total. This indicates cost efficiency of MOKAV’s test generation method. We conclude that researchers and practitioners can employ MOKAV to generate DETs for APR patches. Such tests are highly useful for automated patch assessment, for example for clustering [9].

3.2.3 Conclusion

We make two major contributions to the field of automated patch assessment. First, in LIGHTER, we propose an efficient and lightweight method for employing historical data to estimate the potential of fix templates. In MOKAV, we introduce a novel and efficient technique for generating concrete test data to distinguish between patches. These contributions take a significant step in making automated patch assessment more efficient.

3.3 P3: Making Manual Patch Assessment More Efficient by Facilitating Reasoning about Patch behavior

With all the improvements in patch generation and automated patch assessment, the state-of-the-art APR approaches still cannot guarantee the correctness of their output patches [45]. This necessitates manual patch assessment, which requires a thorough code review on the patches to ensure their correctness. During patch reviewing, the reviewer should understand how the patch modifies the behavior of the program. Does this modification comply with the actual intended behavior? This task is facilitated by various techniques, such as generating natural explanation for the patch [17], guiding patch generation with the developer’s comment on runtime differences between the patch and buggy version [18], and integrating patch review into common development environments [12].

Here we present our contribution for making manual patch assessment more efficient by adding runtime information to the patch in a concise and friendly manner.

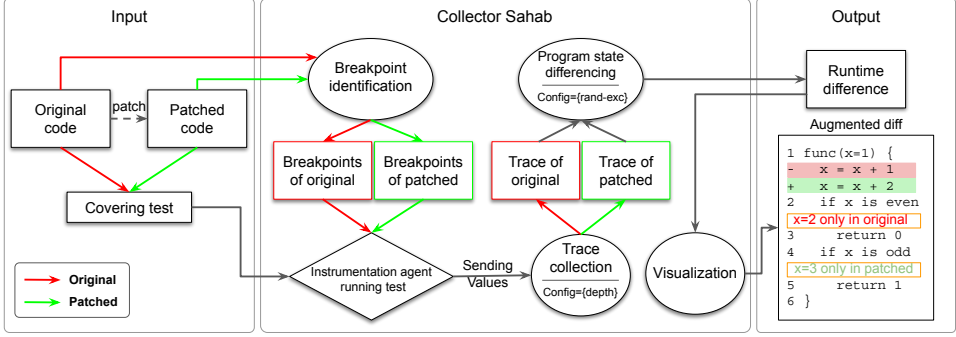


Figure 3.6: Overview of COLLECTOR-SAHAB’s workflow.

3.3.1 Contribution 5: Augmenting Patches with Human-readable Runtime Information

To facilitate the code reviewing process, various forms of explanations may be added to code diffs, such as commit messages [177], code comments, and pull request descriptions [184]. Existing explanation generation methods mostly focus on static information, that is extracted from the code change without running the program [177, 212]. However, runtime information is a great source of concrete and actionable data for explaining code diffs [213, 214].

In this paper, we present a study on using runtime information for explaining code diffs. This is a challenging task because there is an overwhelming number of events and values happening at runtime. This endeavor requires three components: first, a proper algorithm to monitor and select data during the execution of both the original and patched versions of a program; second, an efficient algorithm to extract runtime differences between the collected traces; third, the extracted runtime differences should be integrated into code diffs in a useful manner for code reviewers.

In this contribution, we propose COLLECTOR-SAHAB, a novel system to augment code diffs with runtime information. Given a Java code diff and a test case written in Java, COLLECTOR-SAHAB runs the test on both versions and uses Java bytecode instrumentation to collect *program states* during the execution, where program states consist of values assigned to specific, relevant variables. After collecting program states, COLLECTOR-SAHAB employs an original algorithm to find the variable values that are unique, in the sense that they only occur in one execution of the program under test. The first relevant unique variable value is added to the code diff to obtain an *augmented diff*, that is a diff with both static and dynamic differences. COLLECTOR-SAHAB is the first approach that can provide developers with a concise set of runtime differences, usable for code review.

COLLECTOR-SAHAB's Approach

We design and implement COLLECTOR-SAHAB to generate a concise and useful report of runtime differences caused by a code change. This report puts the extracted execution data into a user interface (UI) to present code diffs. We call this report the *augmented diff* as it augments the code diff with dynamic information.

Figure 3.6 represents an overview of how COLLECTOR-SAHAB works. COLLECTOR-SAHAB gets an original and a patched version of a program as well as a test case to execute them. In the first step, COLLECTOR-SAHAB executes the test on the original and patched versions of the program and compares the program states that occur during these two executions. Then, it selects and reports a set of relevant values that occur in the program states of the original version but are absent in the patched version, and vice versa. This set of values represents the relevant runtime differences between two versions of a program. Finally, COLLECTOR-SAHAB integrates the extracted relevant execution differences into the code diffs typically used in code review. The result is an augmented diff, it enables developers to see runtime differences, together with the code change.

Input: COLLECTOR-SAHAB takes in 3 inputs: the original version of the project, the patched version of the project, and a list of **covering tests**. A covering test is a test that runs at least part of the code that is changed.

Breakpoint Identification: Given the original and patched versions, COLLECTOR-SAHAB finds a mapping between the common lines of the original and patched versions. We call them the *matched lines*. COLLECTOR-SAHAB registers breakpoints at all matched lines in the changed method. We focus on the matched lines so that we can compare the data between the two versions.

Trace Collection: After registering the breakpoints, COLLECTOR-SAHAB runs the covering test on instrumented versions of both original and patched versions and collects their execution traces. The execution trace contains the program states at each execution of each breakpoint. The program state is the set of values assigned to visible variables or returned by methods during the execution of a breakpoint. This includes the values assigned to primitive and non-primitive variables. For non-primitive variables, the values are collected up to a pre-configured *depth*. For example, Listing 3.5 shows that the depth should be greater than or equal to three to make the trace collector collect the value of "student1.supervisor.education.city". With the depth configuration, COLLECTOR-SAHAB enables users to set the level of granularity of runtime differences they want to be reported.

Program State Differencing: After the program states for both original and patched versions are collected, a fast algorithm is employed to compare them.

```

1  Datatypes in Java-like syntax:
2  class Student {
3      String name;
4      Supervisor supervisor;
5  }
6  class Supervisor {
7      String name;
8      Education education;
9  }
10 class Education {
11     String university;
12     String city;
13 }
14
15 An object of the student datatype, serialized in JSON-like syntax:
16 {
17     "student1": {                                depth >= 1
18         "name": "Alice",                        depth = 1
19         "supervisor": {                          depth >= 2
20             "name": "Bob",                      depth = 2
21             "education": {                      depth >= 3
22                 "university": "KTH_Institute",  depth = 3
23                 "city": "Stockholm"            depth = 3
24             }
25         }
26     }
27 }

```

Listing 3.5: A program state is the runtime information associated to some datatype. In this example, the “Student” datatype is shown together with an object of this class “student1” with its content. The student object contains more or less information, depending on the observation depth, in color.

This algorithm first represents each variable value in a program state as a single string. Then, the representations of original variable values are compared against the representations of patched variable values at the corresponding matched line. The variable values that occur only in one version are reported as program state differences. We call a variable value that only occurs during the execution of one of the versions a *unique state value*.

Visualization: COLLECTOR-SAHAB uses the program state differences extracted in the previous step to produce a user-friendly user interface (UI), which we call the **augmented diff**. This UI adds runtime information to the typical code diff view, e.g. that of GitHub. This new graphical component shows the first unique variable value each version. We consider the first unique state value in our GUI for two reasons. First, to avoid overwhelming the code reviewer with too many differences reported. Second, the literature has shown that the first point of execution divergence is the most relevant for code reviewers [215].

1135	-	int j = 4 * n - 1;
1135	+	int j = 4 * (n - 1);
1136	1136	for (int i = 0; i < j; i += 4) {
<div>  COLLECTOR+SAHAB / covering test: testMathpbx02 </div> <div> j=24 only occurs in the patched version. </div>		
<div>  COLLECTOR-SAHAB / covering test: testMathpbx02 </div> <div> j=27 only occurs in the original version. </div>		

Figure 3.7: The augmented diff generated by COLLECTOR-SAHAB for a code diff for Defects4J’s Math-80, showing important runtime information.

Figure 3.7 shows an example of an augmented diff for a code diff for the Math-20 bug. This generated UI clearly indicates that after changing how variable `j` is computed in line 1135, the variable is assigned a new value in the patched version. As shown in Figure 3.7, a code reviewer directly sees that the execution of test `testMathpbx02` reveals that `j=24` only occurs in the patched version and `j=27` only occurs in the original version. By displaying those concrete runtime differences caused by the code change, COLLECTOR-SAHAB helps code reviewers to make an informed decision about the appropriateness of a code diff.

Evaluation of COLLECTOR-SAHAB’s Effectiveness in Detecting Runtime Differences

To assess COLLECTOR-SAHAB’s effectiveness, we run COLLECTOR-SAHAB on a dataset of plausible APR patches, which have a guaranteed runtime difference with the buggy version. For each patch, COLLECTOR-SAHAB looks for program state differences between the buggy and patched versions and augments their code diff. The key metric in this experiment is the number of code diffs for which COLLECTOR-SAHAB augments the diff. To understand the main causes of COLLECTOR-SAHAB failure, we split the remaining code diffs into three groups and count the number of code diffs in each group: 1) code diffs for which no unique state value is detected, 2) code diffs that cause a memory failure, and 3) code diffs that make COLLECTOR-SAHAB exceed the time limit. Moreover, to assess the impact of depth configuration on the results, we run COLLECTOR-SAHAB with four different state depths: `depth=0,1,2,3`. This maximum depth of three has been found empirically, because collecting data for higher depths requires resources beyond our experimental setup without improving effectiveness.

We compare COLLECTOR-SAHAB against DIDIFFFF, the only existing open-source runtime differencing tool for Java. DIDIFFFF runs two versions of a program

Table 3.8: Results of running COLLECTOR-SAHAB on 584 code diffs in our dataset.

Diff Tool	Augmented Diff	No Diff Detected	Memory Failure	Time Limit
DIDIFFFF	377 (64.5%)	207 (35.5%)	—	—
COLLECTOR-SAHAB (depth=0)	543 (93.0%)	32 (4.9%)	9 (1.5%)	0 (0.0%)
COLLECTOR-SAHAB (depth=1)	555 (95.0%)	18 (3.2%)	9 (1.5%)	2 (0.3%)
COLLECTOR-SAHAB (depth=2)	554 (94.8%)	16 (2.8%)	13 (2.2%)	1 (0.2%)
COLLECTOR-SAHAB (depth=3)	544 (93.1%)	22 (3.9%)	18 (3.0%)	0 (0.0%)

and illustrates the difference between the values assigned to primitive variables in a web interface. DIDIFFFF uses SELOGGER [198] to record the list of values each variable of a program holds during an execution. Next, DIDIFFFF compares the two lists of values for each variable access. The result of this comparison is presented to code reviewers in form of a graphical user interface (GUI). This GUI shows the two lists of values assigned to a variable and highlights if there is a difference between these two lists.

Dataset: In this work, we use a dataset of plausible APR patches collected from the DRR dataset [15]. We specifically consider pairs of `<buggy, plausible_patch>` in which the code diff between `buggy` and `plausible_patch` versions lie in a single method. Also, the considered pairs should use Maven for building, as COLLECTOR-SAHAB is implemented to work on Maven projects. The result is a dataset of 584 pairs of Java programs with guaranteed runtime differences.

Experimental Results: Table 3.8 shows the results of running COLLECTOR-SAHAB on 584 code diffs from our benchmark. In each row, “Diff Tool” shows the used tool and its configuration. The second column “Augmented Diff” indicates the number of code diffs for which a runtime difference is detected and the diff is augmented with a unique state value. The remaining three columns explain the failure modes. “No Diff Detected” indicates the number of code diffs for which no unique state value in either of the versions is found. Finally, “Memory Failure” and “Time Limit” show the number of code diffs on which COLLECTOR-SAHAB faces memory and time limit during execution, respectively.

As shown in Table 3.8, COLLECTOR-SAHAB outperforms the state-of-the-art tool DIDIFFFF. The number of code diffs for which COLLECTOR-SAHAB has a runtime difference is higher for all depths, from 93% (543/584) to 95% (555/584) for different depths, while it is 64.5% (377/584) for DIDIFFFF. COLLECTOR-SAHAB detects and reports a runtime difference in notably more cases than DIDIFFFF. Our manual analysis indicates that there are three reasons why COLLECTOR-SAHAB outperforms DIDIFFFF in terms of identifying runtime differences. First, DIDIFFFF only considers differences between values of primitive types, while COLLECTOR-SAHAB also considers differences between the value of non-primitive variables that refer to objects. Second, DIDIFFFF only considers differences between local variable values, while COLLECTOR-SAHAB also detects the difference between the values returned by methods (inside a return statement without an explicit variable).

The third and last reason that COLLECTOR-SAHAB outperforms DIDIFFFF is related to DIDIFFFF not logging the variable values inside inner classes. The latter case is present in our dataset. These three reasons explain why COLLECTOR-SAHAB is able to detect more runtime differences compared to DIDIFFFF.

The column “No Diff Detected” shows the number of code diffs for which COLLECTOR-SAHAB does not find a unique state value. As shown in Table 3.8, generally, the number of these cases decreases when depth increases: the number of code diffs with no diff detected is 32 for depth=0; this number decreases to 16 for depth=2. This indicates that COLLECTOR-SAHAB’s power to look deep inside non-primitive objects adds to its effectiveness in detecting runtime differences. There is an exception for depth=3, where the number of code diffs without a detected diff increases to 22. Per our detailed manual analysis of the logs, the reason behind this is a problem with one of the libraries that COLLECTOR-SAHAB uses. This library ⁴ fails to serialize very large Java object into JSON format. As the size of trace object collected for depth=3 is as large as 10GB in some cases, this leads to COLLECTOR-SAHAB failure to detect a runtime difference. This shows increasing the depth has its own cost and we should carefully select an optimal depth for configuring COLLECTOR-SAHAB in a given setup.

In the current version, COLLECTOR-SAHAB stores all the collected data in RAM during execution and prints the whole trace of the program under test at the end. As the entire trace may be very large, it causes memory failure in some cases. The “Memory Failure” column of Table 3.8 shows that facing a memory limit during COLLECTOR-SAHAB execution is rare. When the depth is set to zero or one, we face memory limit issues only for 1.5% (9/584) code diffs. When depth is set to two or three, the memory limit causes an issue for 2.2% (7/584) and 3.0% (18/584) of code diffs, respectively. There is room for further improvement in this regard, as memory failures can be partially avoided by printing the execution trace step-by-step.

COLLECTOR-SAHAB puts breakpoints at all matched lines, which means the instrumenter collects the program state at many places. This adds an overhead during the runtime and makes the execution take longer than when the code is not instrumented. With respect to time, the “Time Limit” column shows that we rarely face cases where COLLECTOR-SAHAB fails due to time issues. We exceed the time limit only with depth=1 and depth=2 and only for 0.3% (2/584) and 0.2% (1/584) of the code diffs. This means COLLECTOR-SAHAB is fast enough to finish its analysis on most code diffs.

Overall, we notice that COLLECTOR-SAHAB performs the best when the depth is set to one by augmenting the diff for 95% (555/584) of code diffs. Depth=1 outperforms other depths as it provides an accurate balance between looking at details (better than depth=0) and collecting too much data (better than depth=2,3).

⁴<https://github.com/FasterXML/jackson>

Assessing COLLECTOR-SAHAB’s Usefulness with a User Study

To evaluate how software developers assess COLLECTOR-SAHAB’s augmented diffs, we perform a user study. In this study, we ask experienced developers to comment on the quality of COLLECTOR-SAHAB and DIDIFFFF reports. For this experiment, we have two groups of participants: a group of four participants working in industry, and a group of four participants studying at our university. All the participants are experienced in programming with Java language and creating and merging GitHub pull requests.

We present a selected set of commits and their augmentations with both tools to each participant. For each commit, the participant gives a score between 1 and 5 to each augmented code diff in terms of **usefulness**, **clarity**, and **novelty**. Usefulness means whether the report generated by COLLECTOR-SAHAB/DIDIFFFF helps in understanding the changes made to the program. Clarity means how comprehensible the generated report is. Novelty measures how novel is the information provided by COLLECTOR-SAHAB/DIDIFFFF in the sense that it cannot be obtained by looking at the plain code diff. After the participant finishes scoring, we also ask them which tool they prefer overall.

For this experiment we consider eight real-world commits that change a single source code method and cause non-random runtime differences. We split these commits into two groups (industry and students) so that each group gets 4 commits to analyze. This is an appropriate number as it fits well within the one-hour time limit that we set for each participant.

In answer to each question, the participants give a score between 1 and 5 to the augmented code diff. To have an overall assessment of the given scores, we first compute the average score a participant has given to the augmented diffs of each tool in answer to each question. Then, we calculate the median score given by participants in each group (industry/students) to obtain the overall assessment in relation to each criterion.

Experimental Results: Table 3.9 summarizes the results of the user study. As shown in the table, P1-P4 are the participants from industry and P5-P8 are the student participants. For each participant, the table shows the average score that the participant gives to the reports of each tool per each of the considered criterion: “Usefulness”, “Clarity”, and “Novelty”. For each criteria, the “CS” column represents the scores given to COLLECTOR-SAHAB and “DD” represents the scores given to DIDIFFFF. The last column on the right shows the participant preference overall per the discussion at the end of the study.

All the median scores for COLLECTOR-SAHAB are above 3, except for median usefulness score given by students. This indicates the positive attitude of participants about the tool. Regarding the median usefulness score by students, we ask the participants what can be improved to make the augmentation more useful. Per our discussion, they also want to see the type of the variable that is taking a unique value in one version. This is a concrete suggestion for future improvement

Table 3.9: Scores given by each participant to augmented diffs generated by COLLECTOR-SAHAB and DIDIFFFF. “CS” and “DD” represent COLLECTOR-SAHAB and DIDIFFFF, respectively.

Participant ID		Usefulness		Clarity		Novelty		Preference
		CS	DD	CS	DD	CS	DD	
Industry	P1	3.0	2.3	3.3	3.0	4.3	4.0	COLLECTOR-SAHAB
	P2	4.3	2.0	5.0	1.6	4.3	5.0	COLLECTOR-SAHAB
	P3	3.6	2.3	3.6	2.6	2.0	2.3	COLLECTOR-SAHAB
	P4	2.6	1.3	3.3	1.0	4.3	1.0	COLLECTOR-SAHAB
	median	3.3	2.3	3.3	2.1	4.3	3.1	COLLECTOR-SAHAB
Students	P5	2.0	2.6	3.3	3.0	4.0	3.6	COLLECTOR-SAHAB
	P6	3.0	3.3	2.6	3.6	3.6	2.6	DIDIFFFF
	P7	2.6	3.3	3.6	3.3	3.6	4.0	COLLECTOR-SAHAB
	P8	2.0	3.6	3.0	4.3	2.0	3.0	DIDIFFFF
	median	2.3	3.3	3.1	3.4	3.6	3.8	—

of our tool. We also note that median score that industry participants give to COLLECTOR-SAHAB is at least one point higher than the median score they give to DIDIFFFF in terms of each criteria. This indicates that people from industry find COLLECTOR-SAHAB more helpful. Per our discussion with these participants (P1-P4), we realize that they think using COLLECTOR-SAHAB is more practical, as the data it represents is more concise and its UI is integrated into GitHub.

In the live discussion after the scoring task, we also ask the participants what they think about the idea of augmenting code diffs with runtime data. All participants have a very positive view on this idea. The positive view comes from the fact that these tools can “ease the painful process of code review” as P4 says, and “help [us] catch unexpected behavior” as P2 states. This all indicates there is a promising future for code diff augmentation with runtime data.

3.3.2 Conclusion

Our experiments show that our contribution to manual patch assessment, COLLECTOR-SAHAB, suggests a useful method for facilitating the review of APR patches. COLLECTOR-SAHAB is applicable on real-world patches and provides useful, clear, and novel information to code reviewers. By integrating COLLECTOR-SAHAB into APR approaches, we can make the analysis of APR search spaces significantly more efficient.

Conclusion and Future Work

In this thesis, we have investigated the efficiency of exploration and analysis of APR search spaces. In this chapter, we summarize our contributions on this topic and then discuss potential paths for future research.

4.1 Summary

In this thesis, we focus on addressing the APT inefficiencies at its three main steps: patch generation, automated patch assessment, and manual patch assessment.

To make APR patch generation efficient, we propose two tools: SORALD and CIGAR. SORALD is a template-based APR approach for fixing issues reported by the SONARJAVA static analyzer. SORALD takes advantage of the official documentation for SONARJAVA to design highly accurate templates. Each template of SORALD fixes violations of a certain SONARJAVA rule by generating exactly one patch. This means the search space of SORALD is small and accurate, which reduces the cost of automated and manual patch assessment that should be conducted on the search space patches. We run SORALD on 161 popular GitHub Java projects to fix violations of 10 different SONARJAVA rules. Our experiment shows that SORALD successfully fixes 65% (852/1,307) of its target violations, by generating only one patch for each of them. This shows that SORALD is an effective and efficient APR tool.

Our second contribution on efficient APR patch generation is CIGAR. CIGAR is an LLM-based APR tool that employs an iterative approach. At each iteration, CIGAR gives feedback on executing previously generated patches until a plausible patch is generated. After the first plausible patch is generated, CIGAR iteratively prompts the LLM to generate alternative plausible patches. Again, at each iteration, CIGAR summarizes previously generated plausible patches to guide the model toward generating new patches. Besides providing concise feedback on previously patches, CIGAR adopts a reboot strategy. According to the reboot strategy, the feedback loop ends the repair process start from scratch after every

10 iterations. The reboot strategy improves the efficiency by avoiding the model from repeating the same responses. Overall, we show that CIGAR outperforms the state-of-the-art in terms of token cost by 73%. This shows that CIGAR is able to significantly improve the efficiency of patch generation.

We also make two contributions to improve the efficiency of automated patch assessment: LIGHTER and MOKAV. LIGHTER checks APR fix templates against human-made commits to see how many commits follow the modification patterns of each template. An APR fix template that covers many human-made commits has the potential to generate useful patches. We show the applicability of LIGHTER by using it to check templates of eight template-based APR approaches against 7,583 human-made commits. We show that 9.85% (369/7,583) of these commits follow the modification pattern of at least one of the considered fix templates. This proves the applicability of LIGHTER for estimating the potential of fix template search spaces.

MOKAV is our second contribution to automated patch assessment efficiency. MOKAV is an LLM-based tool for generating difference exposing test inputs. Given two versions of a program P & Q, MOKAV iteratively prompts the LLM to generate test inputs that lead to different outputs on P & Q. At each iteration, MOKAV provides feedback on execution results on previously generated tests until a difference exposing test (DET) is generated. We show that MOKAV outperforms baselines by generating DETs on 81.7% (1,255/1,535) of the pairs of Python programs in our dataset. The DETs generated by MOKAV are essential for differentiating between APR patches. For example, these tests can be used for patch clustering that is proven to significantly reduce the time needed for manual patch assessment [9].

Our final contribution concentrates on efficient manual patch assessment. In this contribution, we introduce COLLECTOR-SAHAB, a tool that augments code diffs with runtime information. COLLECTOR-SAHAB runs a test on two versions of a program and identifies their runtime differences at variable and return value level. The detected runtime differences are then added to the code diff to make it easier for the code reviewer to understand behavioral changes. We show that COLLECTOR-SAHAB successfully detects the runtime difference and augments the code diff for 95.0% (555/584) of the code diffs in our dataset. We also find that developers from academia and industry find COLLECTOR-SAHAB augmentations useful, clear, and novel. This all means COLLECTOR-SAHAB improves the efficiency of manual patch assessment by helping developers in understanding behavioral changes.

4.2 Future Work

The LLMs have shown state-of-the-art performance on most software engineering tasks [216]. This observation indicates that to improve the efficiency of state-of-the-art APR components, we have to leverage LLMs in future. Recall that

APR has three main components that impact its efficiency: patch generation, automated patch assessment, and manual patch assessment. We do not see a significant research gap on using LLMs for manual patch assessment, as there is already a large body of work on LLM-based code review [217]. In contrast, we believe there is still a lot of room for improving the efficiency of patch generation and automated patch assessment with LLMs. We propose two paths for future research in this regard.

4.2.1 Execution-free Patch Validation with LLMs

As explained in subsection 1.1.2, the APR patch generation is performed in two steps: patch synthesis and patch validation. Our contribution in CIGAR [44] studies proposes a method for guiding LLMs toward synthesizing APR patches with minimal token cost. For patch validation, the state-of-the-art test-suite based APR approaches simply run the tests against each patch to check if the patch validates. The process of running all tests against each patch in the search space is a significant source of cost in test-suite based APR [218]. We envision using LLMs for validating the patches without execution.

Recently, the researchers are investigating the potential of LLMs for predicting different program execution outputs, such as test coverage, test results. Dhulipala et al. [219] show that an LLM can predict the code coverage of a test with 89% accuracy. The takeaway of these studies is promising, showing that LLMs can understand what happens with the execution of a program, without actually executing it. This finding indicates that LLMs can also be used for patch validation.

Our first proposal for a future study on APR efficiency is employing LLMs to predict the validity of search space patches without executing them. This requires designing, training, and fine-tuning LLMs for predicting code execution results, which is not yet thoroughly studied.

4.2.2 Patch Similarity Assessment with LLMs

As discussed in subsection 1.1.3, a recent technique that significantly improves APR efficiency is patch clustering [9]. In this technique, the patches are grouped into separate clusters based on their similarities. Then, only one representative patch from each cluster has to be manually assessed, which significantly reduces the required human effort. The success of patch clustering depends on accurately finding similarities and differences between APR patches. Our contribution in MOKAV [204] leverages LLMs to generate tests that expose patch differences. We have not yet investigated the ability of LLMs for assessing the similarities between patches.

Researchers have studied the effectiveness of LLMs for finding similar code fragments [220]. This is useful for various software engineering tasks, such as

code clone detection [221]. A strong method for estimating the similarity between different code fragments can also be used for clustering APR patches.

Our second proposal for future work on APR efficiency is leveraging LLMs for clustering APR patches. This can be facilitated by predicting patch similarities. Semantic equivalence prediction is an even more fundamental software task that can directly improve patch clustering. A long term plan can be using LLMs for proving semantic equivalence between patches, which would guarantee the correct patch can be identified with much less manual patch assessment.

References

- [1] J. P. Laboratory. (2024) Nasa’s curiosity mars rover gets a major software upgrade. [Online]. Available: <https://www.nasa.gov/missions/mars-science-laboratory/curiosity-rover/nasas-curiosity-mars-rover-gets-a-major-software-upgrade/>
- [2] F. Cristian, “Correct and robust programs,” *IEEE Transactions on Software Engineering*, no. 2, pp. 163–174, 1984.
- [3] A. Silva, N. Saavedra, and M. Monperrus, “Gitbug-java: A reproducible benchmark of recent java bugs,” in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 118–122.
- [4] M. Zhivich and R. K. Cunningham, “The real cost of software errors,” *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009.
- [5] C. Zhang, J. Yang, D. Yan, S. Yang, and Y. Chen, “Automated breakpoint generation for debugging,” *J. Softw.*, vol. 8, no. 3, pp. 603–616, 2013.
- [6] Z. Chen, “Source code representations of deep learning for program repair,” Ph.D. dissertation, KTH Royal Institute of Technology, 2023.
- [7] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, “Reversible debugging software-quantify the time and cost saved using reversible debuggers,” *University Cambridge: Cambridge, UK*, 2013.
- [8] M. Monperrus, “Automatic software repair: A bibliography,” *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, pp. 1–24, 2018.
- [9] M. Martinez, M. Kechagia, A. Perera, J. Petke, F. Sarro, and A. Aleti, “Test-based patch clustering for automatically-generated patches assessment,” *Empirical Softw. Engg.*, vol. 29, no. 5, jul 2024. [Online]. Available: <https://doi.org/10.1007/s10664-024-10503-2>
- [10] R. Abreu, P. Zoetewij, and A. J. Van Gemund, “On the accuracy of spectrum-based fault localization,” in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

- [11] A. Silva, M. Martinez, B. Danglot, D. Ginelli, and M. Monperrus, “Flacoco: Fault localization for java based on industry-grade coverage,” *arXiv preprint arXiv:2111.12513*, 2021.
- [12] K. Etemadi, N. Harrand, S. Larsén, H. Adzemovic, H. L. Phu, A. Verma, F. Madeiral, D. Wikström, and M. Monperrus, “Sorald: Automatic patch suggestions for sonarqube static analysis violations,” *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 4, pp. 2794–2810, 2022.
- [13] M. Martinez and M. Monperrus, “Ultra-large repair search space with automatically mined templates: The cardumen mode of astor,” in *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*. Springer, 2018, pp. 65–86.
- [14] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, “Nopol: Automatic repair of conditional statement bugs in java programs,” *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.
- [15] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus, “Automated classification of overfitting patches with statically extracted code features,” *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2920–2938, 2021.
- [16] S. S. Bhuiyan, A. Tiwari, Y. Pei, and C. A. Furia, “Ranking plausible patches by historic feature frequencies,” *arXiv preprint arXiv:2407.17240*, 2024.
- [17] D. Sobania, A. Geiger, J. Callan, A. Brownlee, C. Hanna, R. Moussa, M. Z. López, J. Petke, and F. Sarro, “Evaluating explanations for software patches generated by large language models,” in *International Symposium on Search Based Software Engineering*. Springer, 2023, pp. 147–152.
- [18] J. Liang, R. Ji, J. Jiang, S. Zhou, Y. Lou, Y. Xiong, and G. Huang, “Interactive patch filtering as debugging aid,” in *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2021, pp. 239–250.
- [19] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 416–419.
- [20] D. Badampudi, M. Unterkalmsteiner, and R. Britto, “Modern code reviews—survey of literature and practice,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 1–61, 2023.

- [21] N. Jiang, K. Liu, T. Lutellier, and L. Tan, “Impact of code language models on automated program repair,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1430–1442. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ICSE48619.2023.00125>
- [22] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code,” *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01078532/document>
- [23] Y. Yuan and W. Banzhaf, “Arja: Automated repair of java programs via multi-objective genetic programming,” *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.
- [24] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, “Elixir: Effective object-oriented program repair,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 648–659.
- [25] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.
- [26] M. Martinez and M. Monperrus, “Astor: A program repair library for java,” in *Proceedings of the 25th international symposium on software testing and analysis*, 2016, pp. 441–444.
- [27] Z. Qi, F. Long, S. Achour, and M. Rinard, “An analysis of patch plausibility and correctness for generate-and-validate patch generation systems,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 24–36.
- [28] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus, “Npfix: Automatic runtime repair of null pointer exceptions in java,” *arXiv preprint arXiv:1512.07423*, 2015.
- [29] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN international conference on systems, programming, languages, and applications: software for humanity*, 2017, pp. 55–56.
- [30] R. Just, D. Jalali, and M. D. Ernst, “Defects4j: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440.

- [31] H. Ye, “Improving the precision of automatic program repair with machine learning,” Ph.D. dissertation, KTH Royal Institute of Technology, 2023.
- [32] L. Gazzola, D. Micucci, and L. Mariani, “Automatic software repair: A survey,” in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 1219–1219.
- [33] K. Etemadi, N. Tarighat, S. Yadav, M. Martinez, and M. Monperrus, “Estimating the potential of program repair search spaces with commit analysis,” *Journal of Systems and Software*, vol. 188, p. 111263, 2022.
- [34] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, “Bears: An extensible java bug benchmark for automatic program repair studies,” in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 468–478.
- [35] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, “Bugs. jar: A large-scale, diverse dataset of real-world java bugs,” in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 10–13.
- [36] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, “Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.
- [37] Z. Lin, D. Marinov, H. Zhong, Y. Chen, and J. Zhao, “Jacontebe: A benchmark suite of real-world java concurrency bugs (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 178–189.
- [38] N. Saavedra, A. Silva, and M. Monperrus, “Gitbug-actions: Building reproducible bug-fix benchmarks with github actions,” in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, 2024, pp. 1–5.
- [39] J. Xu, Y. Fu, S. H. Tan, and P. He, “Aligning llms for fl-free program repair,” *arXiv preprint arXiv:2404.08877*, 2024.
- [40] L. Peng and C. Mi, “Fault localization of statement coverage based on dnn,” in *2021 International Conference on Electronic Information Technology and Smart Agriculture (ICEITSA)*. IEEE, 2021, pp. 456–463.
- [41] L. Zhang, Z. Li, Y. Feng, Z. Zhang, W. K. Chan, J. Zhang, and Y. Zhou, “Improving fault-localization accuracy by referencing debugging history to alleviate structure bias in code suspiciousness,” *IEEE Transactions on Reliability*, vol. 69, no. 3, pp. 1021–1049, 2020.

- [42] D. Yang, Y. Qi, and X. Mao, “An empirical study on the usage of fault localization in automated program repair,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 504–508.
- [43] S. Yoo, “Evolving human competitive spectra-based fault localisation techniques,” in *Search Based Software Engineering: 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings 4*. Springer, 2012, pp. 244–258.
- [44] D. Hidvégi, K. Etemadi, S. Bobadilla, and M. Monperrus, “Cigar: Cost-efficient program repair with llms,” *arXiv preprint arXiv:2402.06598*, 2024.
- [45] C. S. Xia and L. Zhang, “Keep the conversation going: Fixing 162 out of 337 bugs for \$0.42 each using chatgpt,” *arXiv preprint arXiv:2304.00385*, 2023.
- [46] M. Martinez and M. Monperrus, “Mining software repair models for reasoning on the search space of automated program fixing,” *Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, 2015.
- [47] K. Huang, Z. Xu, S. Yang, H. Sun, X. Li, Z. Yan, and Y. Zhang, “A survey on automated program repair techniques,” *arXiv preprint arXiv:2303.18184*, 2023.
- [48] C. Le Goues, W. Weimer, and S. Forrest, “Representations and operators for improving evolutionary software repair,” in *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, 2012, pp. 959–966.
- [49] D. Ginelli, M. Martinez, L. Mariani, and M. Monperrus, “A comprehensive study of code-removal patches in automated program repair,” *Empirical Software Engineering*, vol. 27, no. 4, p. 97, 2022.
- [50] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, “Automatically finding patches using genetic programming,” in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.
- [51] J. R. Koza, “Genetic programming: on the programming of computers by means of natural selection cambridge,” *MA: MIT Press.[Google Scholar]*, 1992.
- [52] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, “A genetic programming approach to automated software repair,” in *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, 2009, pp. 947–954.
- [53] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, “Automatic program repair with evolutionary computation,” *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

- [54] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, “Bugs as deviant behavior: A general approach to inferring errors in systems code,” *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.
- [55] V. Debroy and W. E. Wong, “Using mutation to automatically suggest fixes for faulty programs,” in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 65–74.
- [56] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, “Automatic error elimination by horizontal code transfer across multiple applications,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2015, pp. 43–54.
- [57] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.
- [58] C.-P. Wong, P. Santiesteban, C. Kästner, and C. Le Goues, “Varfix: balancing edit expressiveness and search effectiveness in automated program repair,” in *Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 354–366.
- [59] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, “Designing better fitness functions for automated program repair,” in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, 2010, pp. 965–972.
- [60] Y. Qi, X. Mao, and Y. Lei, “Efficient automated program repair through fault-recorded testing prioritization,” in *2013 IEEE International Conference on Software Maintenance*. IEEE, 2013, pp. 180–189.
- [61] E. Schulte, S. Forrest, and W. Weimer, “Automated program repair through the evolution of assembly code,” in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 313–316.
- [62] E. Schulte, J. DiLorenzo, W. Weimer, and S. Forrest, “Automated repair of binary and assembly programs for cooperating embedded devices,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 317–328, 2013.
- [63] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, “The strength of random search on automated program repair,” in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 254–265.
- [64] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, “Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset,” *Empirical Software Engineering*, vol. 22, pp. 1936–1964, 2017.

- [65] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, “Context-aware patch generation for better automated program repair,” in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 1–11.
- [66] T. Ji, L. Chen, X. Mao, and X. Yi, “Automated program repair by using similar code containing fix ingredients,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 197–202.
- [67] Y. Wang, Y. Chen, B. Shen, and H. Zhong, “Crsearcher: Searching code database for repairing bugs,” in *Proceedings of the 9th Asia-Pacific Symposium on Internetwork*, 2017, pp. 1–6.
- [68] Q. Xin and S. P. Reiss, “Leveraging syntax-related code for automated program repair,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 660–670.
- [69] Q. Xin and S. Reiss, “Better code search and reuse for better program repair,” in *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. IEEE, 2019, pp. 10–17.
- [70] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, “Shaping program repair space with existing patches and similar code,” in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 298–309.
- [71] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, “A survey of learning-based automated program repair,” *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 2, pp. 1–69, 2023.
- [72] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, “You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems,” in *2019 12th IEEE conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 102–113.
- [73] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, “ifixr: Bug report driven program repair,” in *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2019, pp. 314–325.
- [74] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 802–811.
- [75] S. H. Tan and A. Roychoudhury, “relifix: Automated repair of software regressions,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.

- [76] X. B. D. Le, D. Lo, and C. Le Goues, “History driven program repair,” in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 213–224.
- [77] F. Long, P. Amidon, and M. Rinard, “Automatic inference of code transforms for patch generation,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.
- [78] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, “Mining fix patterns for findbugs violations,” *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.
- [79] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, “Fixminer: Mining relevant fix patterns for automated program repair,” *Empirical Software Engineering*, vol. 25, pp. 1980–2024, 2020.
- [80] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Avatar: Fixing semantic bugs with fix patterns of static analysis violations,” in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 1–12.
- [81] D. Hovemeyer and W. Pugh, “Finding bugs is easy,” *Acm sigplan notices*, vol. 39, no. 12, pp. 92–106, 2004.
- [82] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, “Tbar: Revisiting template-based automated program repair,” in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 31–42.
- [83] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, “Semfix: Program repair via semantic analysis,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 772–781.
- [84] S. Mechtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 448–458.
- [85] —, “Angelix: Scalable multiline program patch synthesis via symbolic analysis,” in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.
- [86] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues, “Sosrepair: Expressive semantic search for real-world program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 10, pp. 2162–2181, 2019.
- [87] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, “Repairing programs with semantic code search (t),” in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 295–306.

- [88] T. Durieux and M. Monperrus, “Dynamoth: dynamic code synthesis for automatic program repair,” in *Proceedings of the 11th International Workshop on Automation of Software Test*, 2016, pp. 85–91.
- [89] J. Bader, A. Scott, M. Pradel, and S. Chandra, “Getafix: learning to fix bugs automatically,” *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 159:1–159:27, 2019. [Online]. Available: <https://doi.org/10.1145/3360585>
- [90] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O’Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, “Moving fast with software verification,” in *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, ser. Lecture Notes in Computer Science, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 9058. Springer, 2015, pp. 3–11. [Online]. Available: https://doi.org/10.1007/978-3-319-17524-9_1
- [91] E. Aftandilian, R. Sauciu, S. Priya, and S. Krishnan, “Building useful program analysis tools using an extensible java compiler,” in *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. IEEE Computer Society, 2012, pp. 14–23. [Online]. Available: <https://doi.org/10.1109/SCAM.2012.28>
- [92] D. Marcilio, C. A. Furia, R. Bonifácio, and G. Pinto, “Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings,” *J. Syst. Softw.*, vol. 168, p. 110671, 2020. [Online]. Available: <https://doi.org/10.1016/j.jss.2020.110671>
- [93] C. S. Xia, Y. Wei, and L. Zhang, “Automated program repair in the era of large pre-trained language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1482–1494.
- [94] M. Marescotti, “Parallelization and modelling techniques for scalable smt-based verification,” Ph.D. dissertation, Università della Svizzera Italiana, 2020.
- [95] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.
- [96] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, “Hoppity: Learning graph transformations to detect and fix bugs in programs,” in *International conference on learning representations (ICLR)*, 2020.
- [97] H. Ye and M. Monperrus, “Iter: Iterative neural repair for multi-location patches,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

- [98] H. Ye, M. Martinez, X. Luo, T. Zhang, and M. Monperrus, “Selfapr: Self-supervised program repair with test execution diagnostics,” in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–13.
- [99] A. Silva, S. Fang, and M. Monperrus, “Repairllama: Efficient representations and fine-tuned adapters for program repair,” *arXiv preprint arXiv:2312.15698*, 2023.
- [100] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, pp. 1–29, 2019.
- [101] N. Kalchbrenner and P. Blunsom, “Recurrent continuous translation models,” in *Proceedings of the 2013 conference on empirical methods in natural language processing*, 2013, pp. 1700–1709.
- [102] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, “Sorting and transforming program repair ingredients via deep learning code similarities,” in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 479–490.
- [103] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, “Coconut: combining context-aware neural translation models using ensemble for program repair,” in *Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis*, 2020, pp. 101–114.
- [104] Y. Li, S. Wang, and T. N. Nguyen, “Dlfix: Context-based code transformation learning for automated program repair,” in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 602–614.
- [105] Y. Tang, L. Zhou, A. Blanco, S. Liu, F. Wei, M. Zhou, and M. Yang, “Grammar-based patches generation for automated program repair,” in *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021, pp. 1300–1305.
- [106] H. Ye, M. Martinez, and M. Monperrus, “Neural program repair with execution-based backpropagation,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1506–1518.
- [107] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, “Codit: Code editing with tree-based neural models,” *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2020.
- [108] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, “A syntax-guided edit decoder for neural program repair,” in *Proceedings of the*

- 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 341–353.
- [109] N. Jiang, T. Lutellier, and L. Tan, “Cure: Code-aware neural machine translation for automatic program repair,” in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1161–1173.
- [110] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, “Improving language understanding by generative pre-training,” *OpenAI Blog*, 2018.
- [111] Q. Zhang, C. Fang, Y. Xie, Y. Ma, W. Sun, and Y. Y. Z. Chen, “A systematic literature review on large language models for automated program repair,” *arXiv preprint arXiv:2405.01466*, 2024.
- [112] A. Vaswani, “Attention is all you need,” *arXiv preprint arXiv:1706.03762*, 2017.
- [113] J. Devlin, “Bert: Pre-training of deep bidirectional transformers for language understanding,” *arXiv preprint arXiv:1810.04805*, 2018.
- [114] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [115] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, “Exploring the limits of transfer learning with a unified text-to-text transformer,” *Journal of machine learning research*, vol. 21, no. 140, pp. 1–67, 2020.
- [116] C. S. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 959–971.
- [117] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, “Codebert: A pre-trained model for programming and natural languages,” *arXiv preprint arXiv:2002.08155*, 2020.
- [118] W. Yuan, Q. Zhang, T. He, C. Fang, N. Q. V. Hung, X. Hao, and H. Yin, “Circle: continual repair across programming languages,” in *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*, 2022, pp. 678–690.
- [119] OpenAI. (2024) Models - openai api. [Online]. Available: <https://platform.openai.com/docs/models>

- [120] J. Li, D. Li, C. Xiong, and S. Hoi, “Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation,” in *International conference on machine learning*. PMLR, 2022, pp. 12 888–12 900.
- [121] Q. Zhang, T. Zhang, J. Zhai, C. Fang, B. Yu, W. Sun, and Z. Chen, “A critical review of large language model on software engineering: An example from chatgpt and automated program repair,” *arXiv preprint arXiv:2310.08879*, 2023.
- [122] R. Paul, M. M. Hossain, M. L. Siddiq, M. Hasan, A. Iqbal, and J. C. S. Santos, “Enhancing automated program repair through fine-tuning and prompt engineering,” 2023.
- [123] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, “Gamma: Revisiting template-based automated program repair via mask prediction,” *arXiv preprint arXiv:2309.09308*, 2023.
- [124] J. Cao, M. Li, M. Wen, and S.-c. Cheung, “A study on prompt design, advantages and limitations of chatgpt for deep learning program repair,” *arXiv preprint arXiv:2304.08191*, 2023.
- [125] N. Nashid, M. Sintaha, and A. Mesbah, “Retrieval-based prompt selection for code-related few-shot learning,” in *Proceedings of the 45th International Conference on Software Engineering (ICSE’23)*, 2023.
- [126] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” *arXiv preprint arXiv:1908.10084*, 2019.
- [127] S. Robertson, H. Zaragoza *et al.*, “The probabilistic relevance framework: Bm25 and beyond,” *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.
- [128] T. Ahmed and P. Devanbu, “Better patching using llm prompting, via self-consistency,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2023, pp. 1742–1746.
- [129] S. Chakraborty and B. Ray, “On multi-modal learning of editing source code,” in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 443–455.
- [130] M. M. A. Haque, W. U. Ahmad, I. Lourentzou, and C. Brown, “Fixeval: Execution-based evaluation of program fixes for programming problems,” in *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. IEEE, 2023, pp. 11–18.

- [131] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, “How effective are neural networks for fixing security vulnerabilities,” *arXiv preprint arXiv:2305.18607*, 2023.
- [132] Q. Zhang, C. Fang, B. Yu, W. Sun, T. Zhang, and Z. Chen, “Pre-trained model-based automated software vulnerability repair: How far are we?” *IEEE Transactions on Dependable and Secure Computing*, 2023.
- [133] E. Mashhadi and H. Hemmati, “Applying codebert for automated program repair of java simple bugs,” in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 505–509.
- [134] R.-M. Karampatsis and C. Sutton, “How often do single-statement bugs occur? the manystubs4j dataset,” in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 573–577.
- [135] A. Zirak and H. Hemmati, “Improving automated program repair with domain adaptation,” *ACM Transactions on Software Engineering and Methodology*, 2022.
- [136] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-t. Yih, L. Zettlemoyer, and M. Lewis, “InCoder: A generative model for code infilling and synthesis,” *arXiv preprint arXiv:2204.05999*, 2022.
- [137] K. Huang, X. Meng, J. Zhang, Y. Liu, W. Wang, S. Li, and Y. Zhang, “An empirical study on fine-tuning large language models of code for automated program repair,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, 2023, pp. 1162–1174.
- [138] D. Sobania, M. Briesch, C. Hanna, and J. Petke, “An analysis of the automatic bug fixing performance of chatgpt,” *arXiv preprint arXiv:2301.08653*, 2023.
- [139] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, “Automated repair of programs from large language models,” in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1469–1481.
- [140] C. Liu, P. Cetin, Y. Patodia, S. Chakraborty, Y. Ding, and B. Ray, “Automated code editing with search-generate-modify,” *arXiv preprint arXiv:2306.06490*, 2023.
- [141] Y. Zhang, Z. Jin, Y. Xing, and G. Li, “Steam: Simulating the interactive behavior of programmers for automatic bug fixing,” *arXiv preprint arXiv:2308.14460*, 2023.
- [142] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, “Agentless: Demystifying llm-based software engineering agents,” *arXiv preprint arXiv:2407.01489*, 2024.

- [143] I. Bouzenia, P. Devanbu, and M. Pradel, “Repairagent: An autonomous, llm-based agent for program repair,” *arXiv preprint arXiv:2403.17134*, 2024.
- [144] B. Berabi, J. He, V. Raychev, and M. Vechev, “Tfix: Learning to fix coding errors with a text-to-text transformer,” in *International Conference on Machine Learning*. PMLR, 2021, pp. 780–791.
- [145] Z. Chen, S. Kommrusch, and M. Monperrus, “Neural transfer learning for repairing security vulnerabilities in c code,” *IEEE Transactions on Software Engineering*, vol. 49, no. 1, pp. 147–165, 2022.
- [146] J. Chi, Y. Qu, T. Liu, Q. Zheng, and H. Yin, “Seqtrans: automatic vulnerability fix via sequence to sequence learning,” *IEEE Transactions on Software Engineering*, vol. 49, no. 2, pp. 564–585, 2022.
- [147] M. Fu, C. Tantithamthavorn, T. Le, V. Nguyen, and D. Phung, “Vulrepair: a t5-based automated software vulnerability repair,” in *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*, 2022, pp. 935–947.
- [148] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, “Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation,” *arXiv preprint arXiv:2109.00859*, 2021.
- [149] N. Wadhwa, J. Pradhan, A. Sonwane, S. P. Sahu, N. Natarajan, A. Kanade, S. Parthasarathy, and S. Rajamani, “Core: Resolving code quality issues using llms,” *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 789–811, 2024.
- [150] J. Zhang, R. Krishna, A. H. Awadallah, and C. Wang, “Ecoassistant: Using llm assistant more affordably and accurately,” *arXiv preprint arXiv:2310.03046*, 2023.
- [151] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, “Is the cure worse than the disease? overfitting in automated program repair,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 532–543.
- [152] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, “On reliability of patch correctness assessment,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 524–535.
- [153] Q. Xin and S. P. Reiss, “Identifying test-suite-overfitted patches through test case generation,” in *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, 2017, pp. 226–236.

- [154] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, “Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system,” *Empirical Software Engineering*, vol. 24, pp. 33–67, 2019.
- [155] Y. Dong, X. Cheng, Y. Yang, L. Zhang, S. Wang, and L. Kong, “A method to identify overfitting program repair patches based on expression tree,” *Science of Computer Programming*, p. 103105, 2024.
- [156] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *29th International Conference on Software Engineering (ICSE’07)*. IEEE, 2007, pp. 75–84.
- [157] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, “A comprehensive study of automatic program repair on the quixbugs benchmark,” *Journal of Systems and Software*, vol. 171, p. 110825, 2021.
- [158] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan, “Better test cases for better automated program repair,” in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, 2017, pp. 831–841.
- [159] X. Gao, S. Mechtaev, and A. Roychoudhury, “Crash-avoiding program repair,” in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.
- [160] L. D’Antoni, R. Samanta, and R. Singh, “Qclose: Program repair with quantitative objectives,” in *International Conference on Computer Aided Verification*. Springer, 2016, pp. 383–401.
- [161] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, “Anti-patterns in search-based program repair,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 727–738.
- [162] B. Lin, S. Wang, M. Wen, and X. Mao, “Context-aware code change embedding for better patch correctness assessment,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–29, 2022.
- [163] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, “Evaluating representation learning of code changes for predicting patch correctness in program repair,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 981–992.
- [164] Q. Zhang, C. Fang, W. Sun, Y. Liu, T. He, X. Hao, and Z. Chen, “Appt: Boosting automated patch correctness prediction via fine-tuning pre-trained models,” *IEEE Transactions on Software Engineering*, 2024.

- [165] A. Z. Yang, S. Kolak, V. J. Hellendoorn, R. Martins, and C. L. Goues, “Revisiting unnaturalness for automated program repair in the era of large language models,” *arXiv preprint arXiv:2404.15236*, 2024.
- [166] T. Le-Cong, D.-M. Luong, X. B. D. Le, D. Lo, N.-H. Tran, B. Quang-Huy, and Q.-T. Huynh, “Invalidator: Automated patch correctness assessment via semantic and syntactic reasoning,” *IEEE Transactions on Software Engineering*, vol. 49, no. 6, pp. 3411–3429, 2023.
- [167] A. Bacchelli and C. Bird, “Expectations, outcomes, and challenges of modern code review,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.
- [168] S. Marwan, N. Lytle, J. J. Williams, and T. Price, “The impact of adding textual explanations to next-step hints in a novice programming environment,” in *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2019, pp. 520–526.
- [169] R. P. Buse and W. Weimer, “Automatically documenting program changes.” in *ASE*, vol. 10, 2010, pp. 33–42.
- [170] M. Linares-Vásquez, L. F. Cortés-Coy, J. Aponte, and D. Poshyvanyk, “ChangescrIBE: A tool for automatically generating commit messages,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 709–712.
- [171] J. Shen, X. Sun, B. Li, H. Yang, and J. Hu, “On automatic summarization of what and why information in source code changes,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2016, pp. 103–112.
- [172] N. Dragan, M. L. Collard, and J. I. Maletic, “Reverse engineering method stereotypes,” in *2006 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 24–34.
- [173] M. Vishalakshi and D. V. Krishnapriya, “Automatic generation of commit messages using natural language processing,” *International Journal of Science and Research (IJSR)*, vol. 5, no. 3, p. 1081–1085, May 2016.
- [174] P. Loyola, E. Marrese-Taylor, and Y. Matsuo, “A neural architecture for generating natural language descriptions from source code changes,” *arXiv preprint arXiv:1704.04856*, 2017.
- [175] P. Loyola, E. Marrese-Taylor, J. Balazs, Y. Matsuo, and F. Satoh, “Content aware source code change description generation,” in *Proceedings of the 11th International Conference on Natural Language Generation*, 2018, pp. 119–128.

- [176] L. Y. Nie, C. Gao, Z. Zhong, W. Lam, Y. Liu, and Z. Xu, “Contextualized code representation learning for commit message generation,” *arXiv preprint arXiv:2007.06934*, 2020.
- [177] S. Jiang, A. Armaly, and C. McMillan, “Automatically generating commit messages from diffs using neural machine translation,” in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.
- [178] K. Etemadi and M. Monperrus, “On the relevance of cross-project learning with nearest neighbours for commit message generation,” in *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 2020, pp. 470–475.
- [179] Q. Liu, Z. Liu, H. Zhu, H. Fan, B. Du, and Y. Qian, “Generating commit messages from diffs using pointer-generator network,” in *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 2019, pp. 299–309.
- [180] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, “Commit message generation for source code changes,” in *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 7 2019, pp. 3975–3981. [Online]. Available: <https://doi.org/10.24963/ijcai.2019/552>
- [181] S. Liu, C. Gao, S. Chen, L. Y. Nie, and Y. Liu, “Atom: Commit message generation based on abstract syntax tree and hybrid ranking,” *arXiv preprint arXiv:1912.02972*, 2019.
- [182] A. Aizawa, “An information-theoretic perspective of tf-idf measures,” *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.
- [183] T. T. Vu, T.-D. Bui, T.-D. Do, T.-T. Nguyen, H. D. Vo, and S. Nguyen, “Automated description generation for software patches,” *arXiv preprint arXiv:2402.03805*, 2024.
- [184] Z. Liu, X. Xia, C. Treude, D. Lo, and S. Li, “Automatic generation of pull request descriptions,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 176–188.
- [185] S. Fang, T. Zhang, Y.-S. Tan, Z. Xu, Z.-X. Yuan, and L.-Z. Meng, “Prhan: Automated pull request description generation based on hybrid attention network,” *Journal of Systems and Software*, p. 111160, 2021.
- [186] P. Mahbub, O. Shuvo, and M. M. Rahman, “Explaining software bugs leveraging code structures in neural machine translation,” *arXiv preprint arXiv:2212.04584*, 2022.

- [187] M. Dias, “Supporting software integration activities with first-class code changes,” Ph.D. dissertation, Laboratoire d’Informatique Fondamentale de Lille, 2015.
- [188] Mergely. (2022) Diff text documents online with mergely, an editor and html5 javascript library. [Online]. Available: <https://www.mergely.com/>
- [189] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.
- [190] B. Fluri, M. Wursch, M. Plnzer, and H. Gall, “Change distilling: Tree differencing for fine-grained source code change extraction,” *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.
- [191] M. J. Decker, M. L. Collard, L. G. Volkert, and J. I. Maletic, “srcdiff: A syntactic differencing approach to improve the understandability of deltas,” *Journal of Software: Evolution and Process*, vol. 32, no. 4, p. e2226, 2020.
- [192] M. Sulír, M. Bačíková, S. Chodarev, and J. Porubán, “Visual augmentation of source code editors: A systematic review,” *Computer Languages, Systems & Structures*, 2018.
- [193] S. Proksch, S. Nadi, S. Amann, and M. Mezini, “Enriching in-ide process information with fine-grained source code history,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 250–260.
- [194] J. Winter, M. Aniche, J. Cito, and A. v. Deursen, “Monitoring-aware ides,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 420–431.
- [195] J. Cito, P. Leitner, M. Rinard, and H. C. Gall, “Interactive production performance feedback in the ide,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 971–981.
- [196] J. Bohnet, S. Voigt, and J. Döllner, “Projecting code changes onto execution traces to support localization of recently introduced bugs,” in *Proceedings of the 2009 ACM symposium on Applied Computing*, 2009, pp. 438–442.
- [197] T. Kanda, K. Shimari, and K. Inoue, “didiff: A viewer for comparing changes in both code and execution traces,” in *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. IEEE, 2022, pp. 528–532.

- [198] K. Shimari, T. Ishio, T. Kanda, and K. Inoue, “Near-omniscient debugging for java using size-limited execution trace,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 398–401.
- [199] S. Urli, Z. Yu, L. Seinturier, and M. Monperrus, “How to design a program repair bot? insights from the repairnator project,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2018, pp. 95–104.
- [200] M. Monperrus, S. Urli, T. Durieux, M. Martinez, B. Baudry, and L. Seinturier, “Repairnator patches programs automatically,” *Ubiquity*, vol. 2019, no. July, pp. 1–12, 2019.
- [201] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, “Sapfix: Automated end-to-end repair at scale,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.
- [202] A. Carvalho, W. Luz, D. Marcílio, R. Bonifácio, G. Pinto, and E. D. Canedo, “C-3pr: A bot for fixing static analysis violations via pull requests,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 161–171.
- [203] D. Serban, B. Golsteijn, R. Holdorp, and A. Serebrenik, “Saw-bot: Proposing fixes for static analysis warnings with github suggestions,” in *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 2021, pp. 26–30.
- [204] K. Etemadi, B. Mohammadi, Z. Su, and M. Monperrus, “Mokav: Execution-driven differential testing with llms,” *arXiv preprint arXiv:2406.10375*, 2024.
- [205] M. Martinez, W. Weimer, and M. 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*, 2014, pp. 492–495.
- [206] M. Martinez and M. Monperrus, “Coming: a tool for mining change pattern instances from git commits,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 79–82.
- [207] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, “Towards an automated approach for bug fix pattern detection,” in *VEM ’18 - Proceedings of the VI Workshop on Software Visualization, Evolution and Maintenance*, São Carlos, Brazil, Sep. 2018. [Online]. Available: <https://hal.archives-ouvertes.fr/hal-01851813>

- [208] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, “An empirical evaluation of using large language models for automated unit test generation,” *IEEE Transactions on Software Engineering*, 2023.
- [209] K. Etemadi, A. Sharma, F. Madeiral, and M. Monperrus, “Augmenting diffs with runtime information,” *IEEE Transactions on Software Engineering*, 2023.
- [210] S. Lukasczyk and G. Fraser, “Pynguin: Automated unit test generation for python,” in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.
- [211] T.-O. Li, W. Zong, Y. Wang, H. Tian, Y. Wang, S.-C. Cheung, and J. Kramer, “Nuances are the key: Unlocking chatgpt to find failure-inducing tests with differential prompting,” in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 14–26.
- [212] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, “Atom: Commit message generation based on abstract syntax tree and hybrid ranking,” *IEEE Transactions on Software Engineering*, 2020.
- [213] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, “Codehint: Dynamic and interactive synthesis of code snippets,” in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 653–663.
- [214] A. Gosain and G. Sharma, “A survey of dynamic program analysis techniques and tools,” in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014: Volume 1*. Springer, 2015, pp. 113–122.
- [215] Y. Zhang, K. Rodrigues, Y. Luo, M. Stumm, and D. Yuan, “The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 131–146.
- [216] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, “Large language models for software engineering: Survey and open problems,” in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023, pp. 31–53.
- [217] Y. Yu, G. Rong, H. Shen, H. Zhang, D. Shao, M. Wang, Z. Wei, Y. Xu, and J. Wang, “Fine-tuning large language models to improve accuracy and comprehensibility of automated code review,” *ACM Transactions on Software Engineering and Methodology*, 2024.

- [218] Y.-A. Xiao, C. Yang, B. Wang, and Y. Xiong, “Accelerating patch validation for program repair with interception-based execution scheduling,” *IEEE Transactions on Software Engineering*, 2024.
- [219] H. Dhulipala, A. Yadavally, and T. N. Nguyen, “Planning to guide llm for code coverage prediction,” in *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, 2024, pp. 24–34.
- [220] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, and X. Huang, “Towards understanding the capability of large language models on code clone detection: a survey,” *arXiv preprint arXiv:2308.01191*, 2023.
- [221] Z. Zhang and T. Saber, “Assessing the code clone detection capability of large language models,” in *2024 4th International Conference on Code Quality (ICCQ)*. IEEE, 2024, pp. 75–83.

Part B