Explaining Software Faults in Source Code
Francisco José Torres Ribeiro
UMinho | 2024

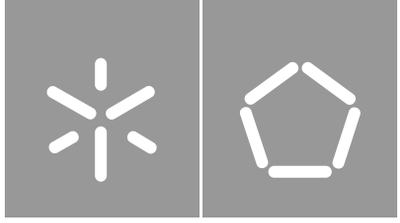
Universidade do Minho
Escola de Engenharia

Francisco José Torres Ribeiro

Explaining Software Faults in Source Code

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

maio de 2024



Universidade do Minho
Escola de Engenharia

Francisco José Torres Ribeiro

Explaining Software Faults in Source Code

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

Trabalho efetuado sob a orientação do
Professor Doutor João Alexandre Baptista Vieira Saraiva
e do
Professor Doutor Rui Filipe Lima Maranhão de Abreu

maio de 2024

Acknowledgements

This thesis was only possible due to the support of many parts. Here, I wish to convey my deepest gratitude. I hope these brief words are enough to express how dearly I hold them.

I would like to start by thanking my excellent supervisors João Saraiva and Rui Maranhão. Their guidance throughout this journey and their support to keep going were crucial to the conclusion of this work. Thank you to both of you!

A very important part of this endeavor happened at NII in Tokyo. There, Professor Kanae Tsushima played a vital role supervising my two internships. Her expertise greatly contributed to the quality of the successful work we developed. Moreover, she showed to be readily available to help with any difficulties I might face during my stay so far away from home. A big thank you!

My friends Rui Rua and José Nuno made sure the PhD journey was a fun one. Surely, without them, the car rides wouldn't have been as nice as they were. And we even got to see the inside of a typical Japanese house. While we are at it, a special thank you to Alex, Zhao (who comes up with the best fun plans), and Zihan, who were a vital part of the social side of my stay in Japan. Also, a special thank you to my friend Walter Lucas, one of the friendliest people I know that contributed to many nice moments. My childhood friend Pedro Campos also had a very important contribution to this work: the car rides to Porto helped me get in the zone many times. André Teixeira, even though he is an engineer, fills in the important spot of someone who I can talk to about things other than informatics. Daniel Cruz plays the other side of that and is a great companion for any plan. You know I cherish our moments together. Obrigado, amigos!

Of course, I also need to thank my girlfriend Xana. Her love and care make me feel extremely fortunate for having her in my life. I suspect that is behind her "promotion" to fiancée.

Finally, my parents get a big thank you for putting up with me. They are always ready with their endless support for everything I set myself to do in life. My brother always motivates me and makes me believe I can get over any struggles. My grandmother is always making sure I am okay no matter what. My dog, Bill, is always ready to greet me with his unending energy and be by my side no matter the circumstances. From the bottom of my heart, thank you.

This work was partially funded by FCT — Foundation for Science and Technology — with PhD scholarship reference SFRH/BD/144938/2019 and within project UIDP/50014/2020. Additionally, this work also benefited from funding provided by JSPS KAKENHI-JP19K20248, ERASMUS+ Programme of the European Union within ERASMUS+ project No. 2020-1-PT01-KA203-078646, COST — European Cooperation in Science and Technology Organisation — within ICT COST Action CA19135: CERCIRAS — Connecting Education and Research Communities for an Innovative Resource Aware Society.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Resumo

Explicar Falhas de Software no Código Fonte

A remoção de falhas, ou bugs, de software é chamada de depuração e é amplamente reconhecida como a tarefa mais árdua no desenvolvimento de software. Envolve deteção, localização e reparação. Enquanto os dois primeiros passos foram amplamente automatizados, a reparação tem sido menos priorizada devido aos desafios na geração de código que aborda vários tipos de bugs e imita estilos de escrita semelhantes aos humanos.

Acreditamos firmemente na melhoria da automatização da fase de reparação, ecoando DeMillo: "os programadores criam programas que estão quase corretos." Esta tese foca-se em aproveitar informação contextual do código-fonte para melhorar a reparação automática de programas (APR), reconhecendo a ligação entre a localização de falhas e a reparação e propondo abordagens para integrar grandes modelos de linguagem (LLMs).

Primeiro, apresentamos o Morpheus, uma técnica de inferência de operadores de mutação que identifica o *onde* e o *porquê* da ocorrência de falhas em software, associando mudanças na árvore do programa com padrões de operadores de mutação. Demonstramos eficiência e eficácia superiores em comparação com métodos tradicionais.

Segundo, utilizando o modelo generativo CodeGPT, abordamos APR como um desafio de *code completion*, gerando linhas de correção candidatas, com base num ficheiro defeituoso e número de linha, através de um processo de várias etapas que envolve *parsing* do código, geração de sequências de tokens, restrição em caracteres de sintaxe e alinhamento de sequências.

Terceiro, propomos o Mentat, uma técnica que utiliza o GPT-3 para reparação automática de erros de tipo através do *bypass* de sistemas de tipos para localizar expressões candidatas. O Mentat supera outras técnicas do estado da arte e permite análise escalável de programas através de uma validação totalmente automatizada, algo raramente feito em investigação.

Coletivamente, estas contribuições abrem caminho para demonstrar como a integração estratégica de LLMs pode superar as limitações inerentes em APR.

Palavras-chave: Compreensão de Programas, Geração de Código, Grandes Modelos de Linguagem, Localização de Falhas, Reparação Automática de Programas

Abstract

Explaining Software Faults in Source Code

Removing faults, or bugs, from software is called debugging and is widely recognized as the most arduous task in software development. It involves detection, localization, and repair. While the first two steps have been largely automated, repair has been less prioritized due to challenges in generating code that addresses various bug types and mimics human-like writing styles.

We firmly believe in improving the automation of the repair phase, echoing DeMillo: "developers create programs that are almost correct." This thesis focuses on leveraging contextual information from source code to enhance automated program repair (APR). It acknowledges the link between fault localization and repair and proposes approaches to integrate large language models (LLMs).

First, we introduce Morpheus, a mutation operator inference technique that identifies where and why software faults occur by associating changes in the program's tree with mutation operator patterns. We demonstrate superior efficiency and effectiveness compared to traditional methods.

Second, using the generative model CodeGPT, we approach APR by treating it as a code completion challenge, generating candidate patch lines based on a given buggy file and line number through a multi-step process involving parsing the code, generating token sequences, constraining syntax characters, and sequence alignment.

Third, we propose Mentat, a technique that utilizes GPT-3 for automated repair of type errors by bypassing type systems to locate candidate expressions. Mentat surpasses other state-of-the-art techniques and enables scalable program analysis through a fully automated validation, something rarely done in research.

Collectively, these contributions pave the way to showcase how strategically integrating LLMs can overcome inherent limitations in APR.

Keywords: Automated Program Repair, Code Generation, Fault Localization, Large Language Models, Program Comprehension

Contents

Acronyms

AI Artificial Intelligence

APR Automated Program Repair

AST Abstract Syntax Tree

BPE Byte-Pair Encoding

DL Deep Learning

FL Fault Localization

GP Genetic Programming

GPT Generative Pre-trained Transformer

LLM Large Language Model

LSP Language Server Protocol

MBFL Model-Based Fault Localization

MBSD Model-Based Software Debugging

MLM Masked Language Modeling

NLP Natural Language Processing

NLU Natural Language Understanding

NLG Natural Language Generation

NMT Neural Machine Translation

PL Programming Language

QR Qualitative Reasoning

Q-SFL Qualitative Spectrum-based Fault Localization

RNN Recurrent Neural Network

SFL Spectrum-based Fault Localization

List of Figures

List of Tables

1 Introduction

Computer programs are vital to the correct operation of a wide range of critical scenarios. They are a driving force in the field of space exploration controlling sophisticated rovers on the Moon and Mars; they oversee transportation systems ensuring quick and safe mobility in the busiest urban centers; and they contribute to healthcare by enabling life-threatening diagnostics.

Computer programs are also an integral part of our daily lives. They enable mobile apps to attend our everyday conveniences assisting our navigation and reminding us of our daily schedules; they make our workplaces more efficient through the automation of data processing; and they constantly monitor our cars assuring we can react on time by warning about potential collisions and correcting any issues with wheels or axis spins that may be out of sync.

Initially, a "computer program" referred to a set of instructions for a specific task. As computers advanced, the simpler term "program" became synonymous. The terminology progressed further, and the term "software" emerged, encompassing the various ways we employ computer instructions. The terms "computer program," "program," and "software" have become interchangeable. This linguistic progression reflects the seamless integration of computers into society. By now, we can safely state that:

Software. Is. Everywhere.

Figure 1.1: The text so far interpreted by DALL·E 3 — an AI image generator[1]

[1] As we move forward in this thesis, this mention of AI will become especially relevant.

But, where does software come from? Developers create software. That is, developers, or programmers, are the people that implement software. The way they do this is by writing code that conveys what the programs should do.

However, despite the best efforts from developers, programs are not immune to errors. Just as humans are prone to making mistakes, programmers can inadvertently introduce bugs into their code. But, what is a bug?

A bug can be defined as a flaw or mistake in the code of a program that leads to incorrect or unintended outcomes, i.e. errors.

These errors can cause unexpected behavior, system failures, and security vulnerabilities. Ultimately, this can result in catastrophic consequences and, because of that, some bugs have made it into history:

- **Therac-25:** A software bug in a medical radiation therapy machine led to patient overdoses, causing serious health complications and fatalities.
- **Ariane 5:** A rocket launch failure occurred due to a software bug in the guidance system, leading to a catastrophic explosion.
- **Patriot Missile Failure:** During the Gulf War, a software bug in the Patriot missile defense system caused it to miscalculate the time, resulting in the failure to intercept an incoming Scud missile and causing casualties.
- **Shellshock:** A critical vulnerability in the Bash shell — a tool that interprets commands and helps users interact with computers — allowed attackers to execute harmful commands, posing significant security risks.
- **Heartbleed:** A flaw in OpenSSL — which acts as a digital lock that keeps information secure — exposed sensitive data, compromising the security of numerous websites and online services.
- **Spectre and Meltdown:** These vulnerabilities in computer processors allowed unauthorized access to sensitive information, affecting a wide range of devices and systems worldwide.

While developers employ various techniques to minimize the occurrence of bugs, it is practically impossible to eliminate them entirely. Indeed, software may very well be released with unknown bugs (Liblit et al., 2005). Given this issue, established companies like Mozilla (Moz,

2017) and Google (Goo, 2021) have bug bounty programs, providing rewards to external developers that detect and fix bugs.

Even more surprisingly, software can be distributed with known bugs! (Anvik et al., 2005; Tian et al., 2013; Natella et al., 2016) Think about it, the existence of bugs is recognized to be so frequent that entities consciously choose to release their software even knowing beforehand that some problems still exist. The effort needed to locate and fix all detected bugs can be so overwhelming, that the preferred option is to publish software before addressing every bug. Granted, these bugs cannot be so severe as to compromise a software's primary functionality.

In 2002, a study revealed that software defects alone incurred an annual expense of $60 billion for the United States economy (Tassey, 2002). According to a 2013 research conducted by Cambridge University, global software bugs were estimated to result in an annual cost of $316 billion for the industry (Cam, 2013). A parallel scenario unfolded in the global economy when a spreadsheet error in a study by economists Reinhart and Rogoff distorted the impact of high debt ratios on economic growth. Many politicians publicly cited this study as a justification for austerity measures before the error was discovered (Gua, 2013).

As a consequence, **detecting, locating, and repairing** bugs are topics of the utmost importance to ensure the reliability and robustness of software systems:

- **Bug Detection:** This involves recognizing the presence of errors. Test cases play a key role in signaling these issues.
- **Fault Localization (FL):** It refers to identifying where the bugs occur within the source code. That is, the location where the bug resides.
- **Program Repair:** This focuses on modifying the program in order to correct its behavior while minimizing the introduction of new bugs.

In a regular situation, fixing bugs involves going through these steps one after the other, and we call this whole process **debugging**. Estimating the precise cost of debugging is challenging, given the fluctuating effort required for each step. Despite that difficulty, one thing is sure: the cost is significant (Vessey, 1985). Research suggests debugging tasks constitute 50% to 75% of total development costs (Hailpern and Santhanam, 2002), with manual debugging and maintenance often consuming 80% of a project's resources (seacord et al., 2003).

In response to the real-world demand for a more effective debugging process, the commu-

nity has looked for techniques to optimize it. The use of automated techniques in debugging not only meets the demand for improved effectiveness but also enables greater efficiency. As software systems become more complex, the integration of automated debugging tools becomes increasingly essential to meet the demands of modern development practices.

It is in this context that generative AI offers a promising avenue, presenting a potential solution to streamline and improve the overall efficiency of the debugging process. Generative AI refers to a class of computational techniques that have the capability to generate new content, be it text, images, or other data, based on patterns learned from vast amounts of existing information[2]. Specifically, we will focus on **large language models** (LLMs), a subset of generative AI. LLMs are powerful systems trained on extensive textual data, enabling them to **understand and generate** human-like language. This positions them as valuable tools for assisting the debugging process.

In this thesis, our main aim is to explore ways to make fixing computer programs easier and more effective. To do this, we will primarily target an area called **automated program repair** (APR). We will be delving into the world of automated techniques that help find and repair mistakes in source code. We will focus on pinpointing not only where things went wrong but also why they went wrong. We will see how understanding the structure of computer programs can help with this. Imagine it like doctors for source code — our goal is to create smarter tools that can find and fix problems on their own, making the process of building and maintaining software smoother and more reliable.

1.1 Bug Detection

One crucial approach to identifying bugs is through the use of test cases. A test case is a specific input or set of inputs along with expected outcomes, designed to assess the behavior and correctness of a program. By running test cases, developers can determine whether their program behaves as expected and if bugs are present.

While test cases are a crucial tool for identifying bugs in software, it's important to note that tests do not actually prove the correctness of a program. Instead, they focus on uncovering errors or unexpected behaviors that might exist in the code. This distinction is essential because there

[2] Hopefully, it now makes sense that we placed Figure 1.1 earlier.

are formal methods and techniques that aim to mathematically prove program correctness. However, these formal approaches often require complex specifications and are not always feasible for every type of software.

Formal methods involve mathematical logic and rigorously defining the behavior of a program in a way that allows for verification of desired properties. While formal proofs can provide a higher level of confidence in the correctness of a program, they come with their own challenges and limitations.

On the other hand, tests provide a practical means of assessing how a program behaves under various conditions. They are designed to simulate real-world scenarios and input variations, exposing bugs and deviations from expected outcomes. However, it's important to acknowledge that tests are not infallible themselves. Just as a formal proof may fail if it is built upon incorrect assumptions (Smith, 1985; Fonseca et al., 2017), tests can only detect issues that are within the scope of the test cases themselves (Fraser and Arcuri, 2011; Holler et al., 2012). If a test case is flawed or doesn't cover a specific aspect of the program, it might not uncover certain bugs.

Tests do provide valuable insights into a program's behavior and help improve its reliability, but they don't guarantee absolute correctness. This is why a combination of testing, code reviews, formal methods, and other software engineering practices are often used together to ensure the quality and dependability of software systems. In essence, both tests and formal proofs have their strengths and limitations, and their effectiveness depends on how well they are applied to the unique challenges of a particular software project.

1.2 Fault Localization

Fault localization (FL) is a critical aspect of the software development process that focuses on identifying the location of bugs within a program's source code. This process is crucial for effective and efficient bug fixing, thus improving the overall reliability of software.

Imagine you're searching for a needle in a haystack. In this analogy, fault localization is the process of narrowing down the area of the haystack where the needle might be located. Instead of sifting through the entire codebase, developers employ various techniques and tools to narrow down the possible locations of errors, making the debugging process more efficient.

Spectrum-based Fault Localization (SFL) utilizes program spectra as a window into program behavior. It distinguishes between normal and problematic scenarios by abstracting behavior into spectra, providing a high-level overview of component contributions. In similarity-based SFL, components behaving like failing tests indicate potential issues, identified through similarity coefficients. On the other hand, reasoning-based SFL, leveraging Bayesian reasoning, addresses both single and multiple, intermittent problems, enhancing fault localization for complex scenarios. Note, however, that single fault scenarios are more prevalent in real-world projects (Perez et al., 2017). Intermittent faults occur sporadically, posing challenges in detection due to their unpredictable nature. In that sense, Q-SFL is a more advanced fault localization method that uses qualitative reasoning (QR) to partition system components, enriching diagnostic insights. This approach considers qualitative properties as distinct components, such as specific values for parameters, recording their involvement and ranking based on similarity to failures.

In this thesis, we depart from the more conventional fault localization methods, particularly aligning with the innovative perspective of Q-SFL. Our approach ventures beyond traditional boundaries by incorporating concepts not commonly associated with fault localization, such as program mutation. This unconventional strategy aims to enhance diagnostic capabilities, introducing novel dimensions to fault comprehension. Moreover, our methodology embraces well-established language engineering techniques which are robustly implemented in tools like parsers and compilers.

This stems from an interesting observation: certain tasks that software tools perform during the development process inadvertently aid in fault localization. Parsers, for instance, are tools that analyze the structure of code to ensure it adheres to the syntax rules of the programming language. While their primary purpose is to validate the correctness of the code's syntax, they also play a role in identifying potential issues or inconsistencies that could lead to errors.

Similarly, type checking and type inference are mechanisms used to ensure that variables and expressions within the code are used in ways that are consistent with their data types. These mechanisms help catch errors related to incompatible data types, but they also provide valuable hints about where potential issues might lie in the code.

By combining the unconventional with the tried-and-true, we forge a path toward a more comprehensive fault localization paradigm.

The hints provided by fault localization significantly enhance and, in some cases, enable the automated repair process, by offering precise guidance on where to focus corrective efforts within the codebase. This is especially significant when considering LLMs, as they heavily rely on the context provided in their input to generate more accurate and precise code solutions. By enhancing prompts with contextual information and hints extracted from fault localization, these models are empowered to better understand the intricacies of the codebase and subsequently generate code solutions that are not only syntactically correct but also contextually appropriate.

1.3 Automated Program Repair

Automated program repair refers to the use of computational techniques and algorithms to automatically fix bugs in software. This emerging field combines concepts from software engineering and program analysis to develop tools that can autonomously generate patches or fixes for identified bugs.

Continuing with the needle in the haystack analogy, if fault localization identifies potential bug locations, APR operates like a precise robotic hand equipped with a versatile toolkit, similar to a screwdriver with interchangeable heads. It systematically experiments with different "screwdriver heads" to discern the most effective one for handling the needle. Alternatively, it can intelligently analyze the suspicious area, selecting the optimal strategy to approach and handle the needle.

Much of the pioneering work in APR has revolved around evolutionary repair, leveraging genetic programming to evolve incorrect programs into their correct counterparts. This approach draws inspiration from biological evolution, treating programs as individuals in a population that systematically reproduce and may undergo mutations. However, despite their flexibility and intuitiveness, many evolutionary repair techniques face a challenge known as overfitting. Overfitting leads to overly specific fixes that address only a part of the verified problem, lacking generalization to other similar scenarios.

Moreover, to address the risk of generating nonsensical patches, some approaches rely on repair templates, representing common methods for addressing prevalent types of bugs. Constraint-based approaches, on the other hand, analyze information from test executions to create constraints fed to a solver for patch generation.

Some of the focus of this thesis intersects with the usage of mutation operators for automated bug fixing. These techniques leverage standard mutation operators documented in the literature and their combination in the form of higher-order mutants.

Let's take a moment to emphasize a core theme in this thesis. As we progress in our discussion, our primary focus becomes apparent: the integration of deep learning (DL) into APR. In the beginning, we labeled APR as an "emerging field". Despite this, APR has quickly matured into an area with impactful contributions, already allowing us to distinguish between traditional approaches — marked by experimenting with various program variants to assess effectiveness — and cutting-edge endeavors that leverage DL.

Despite being a more recent development, DL-based approaches have rapidly evolved, refining their framework for APR in a comparatively short time compared to traditional techniques. Initially perceived as a translation task from buggy code to fixed code, akin to translating between languages, these approaches faced challenges. The advantages offered by pre-trained language models in natural language tasks prompted the software engineering community to explore the utilization of LLMs for automatic repair, addressing these challenges head-on.

Ultimately, the primary goal of APR is to streamline the bug fixing process and reduce the manual effort required from developers. By automating bug repair, developers can save time and resources, enabling them to focus on other critical aspects of software development. Moreover, APR techniques can provide rapid bug fixes, minimizing the impact of bugs on software systems and reducing the time it takes to deploy bug-free updates. This has tremendous implications. One might think that by automating away this laborious task, developers can simply focus on writing more code for the sake of increasing productivity. While this is a reasonable and worthy objective, the sole focus on this benefit is shortsighted. Bug fixing can be as tedious as it is enduring, if not more. Therefore, ridding developers from the burden of fully manual bug fixing opens up opportunities to enhance the quality of their work. Consequently, developers can shift their attention to more stimulating and rewarding tasks. This way, there is more space for critical thinking and imagination, improving the creative process in software development. Freedom and availability to focus on these aspects allow developers to take pride in their work and enjoy it more. Ultimately, programmers will deliver higher quality software. This emphasis on quality over quantity leads to more innovative software.

In the context of APR, the main contributions of this thesis lie in understanding how to best in-

tegrate LLMs into APR tasks. This exploration delves into comprehending the mechanisms behind these models and optimizing their utilization to enhance automated program repair, positioning it as an integral component of the software development process.

1.4 Large Language Models

While this thesis doesn't delve into building or fine-tuning large language models, it is imperative to acknowledge their crucial role in the majority of the contributions here. Here, "large" refers to both the amount of data the models have been trained on and the model's size. When we talk about the "size" of a model, it means the number of internal parameters, called weights, it has. Think of these weights like the gears in a machine. Having more of them allows the model to understand and work with information in a more complex way. As such, a larger model size often means a more powerful system.

Before making language models larger, it is often useful to extrapolate the performance of smaller models to predict some of the effects. However, an intriguing phenomenon, known as emerging properties, comes into play. These are abilities that manifest in larger models but are absent in smaller ones. These emergent properties cannot be predicted because they are not purely a function of scale. That is, emerging properties are qualitative in nature and, therefore, unforeseen by merely studying quantitative changes (Wei et al., 2022).

GPT-3[3] popularized few-shot prompting: the model is provided with a few input-output examples before performing the task on a new example. Under this setting, a significant amount of emerging abilities have been observed. One example is arithmetic reasoning, for which GPT-3 and LaMDA (Thoppilan et al., 2022) exhibit near-zero performance during a significant amount of training effort, but experience a sudden improvement, transforming from relatively weak to highly proficient at the given task.

Indeed, other factors have also played a crucial role in the recent success of LLMs. The introduction of the Transformers architecture (Vaswani et al., 2017) has sped up training times and improved the ability to learn from long sequences, allowing models to better understand

[3]GPT stands for generative pre-trained transformer and is a type of LLM. While OpenAI has a numbered "GPT-n" series, the term "GPT" is not exclusive to the company. If we wish to be precise, we should refer to them as OpenAI's GPT models, although we will loosen this rule throughout this thesis, as is common.

connections between distant words in a sentence. This last point is particularly significant in programming languages because related code elements are more likely to be dispersed compared to natural languages. Therefore, in the context of programming, the capability to establish essential relationships between distant sections is critical.

Traditional APR typically employs intelligent design and domain-specific knowledge — top-down. While many traditional static analysis techniques are effective in such scenarios, they face limitations, especially when dealing with intricate problems. LLMs introduce a transformative perspective, implicitly resolving various types of issues without adhering to predetermined patterns — bottom-up. This shift is crucial, as LLMs excel not only in common issues but also in addressing intricate problems related to program logic and flawed reasoning, providing a more comprehensive approach for APR.

Recent contributions leveraging LLMs for APR cover a range of creative methods, from AlphaRepair (Xia and Zhang, 2022) — which makes direct use of CodeBERT (Feng et al., 2020) to complete missing code — to Tare (Zhu et al., 2023), which teaches the LLM to understand and respect different data types used in the code.

The success of LLMs in APR is dependent on the quality of input prompts. To yield accurate and contextually relevant results, considerable effort must be dedicated to constructing informative prompts that encapsulate the nuances and specifics of the buggy code. The correlation between good-quality input and output highlights the importance of meticulously crafting prompts with relevant context, ensuring that the generated suggestions align with the code's structure and constraints.

In this thesis, we will explore methods to blend LLMs into APR. This will involve extracting valuable insights from buggy code to construct effective input prompts and adjusting LLM-generated output to seamlessly fit it into the programs undergoing repair.

1.5 Problem Statement

Let us take this chance to precisely establish our focus: we exclusively delve into the realm of software bugs. As such, we exclude hardware faults. To refine our scope further, our primary emphasis lies in behavioral repair — the targeted modification of source code to rectify a

program's behavior. Note that these efforts are distinct from addressing a program's runtime state. Throughout this work, we will commonly refer to these software bugs simply as "bugs." While bugs can potentially lead to catastrophic consequences, it's essential to recognize bugs will inevitably happen. We believe this justifies our efforts to fix them. Complete prevention is, ultimately, futile. Therefore, we must do our best to repair bugs as soon as they are detected in the wild or, even better, before they have a chance to cause any trouble in the real-world.

As previously noted, the process of eliminating bugs is called debugging and is the most costly step in software development with conservative estimates accounting for at least 50% of the effort (Vessey, 1985; Hailpern and Santhanam, 2002; Zeller, 2009; Rößler et al., 2013; Wang et al., 2021; Lampel et al., 2021; Steinhöfel and Zeller, 2022; Dutra et al., 2023; Eberlein et al., 2023).

Simultaneously, the escalating trend of software shipping with more bugs has a significant implication: a substantial portion of the debugging effort extends into the maintenance phase (Liblit et al., 2005). Indeed, developer teams, constrained by resources and time, grapple with addressing every bug before releasing software (Anvik et al., 2005). Now recall that software maintenance accounts for between 80% to 90% of the cost of a software project (seacord et al., 2003). As bug repair becomes increasingly prevalent during this phase, the cumulative effect can be overwhelming (Erlikh, 2000; Ramamoorthy and Tsai, 1996; Le Goues et al., 2012b; Yan et al., 2023). This trend is reinforced by the ongoing growth in software deployment, accompanied by a subsequent influx of bug reports (Anvik et al., 2006).

Quoting Dr. Werner Vogels "the cost to build is dwarfed by the cost of operating your applications" — as Figure 1.2 shows.

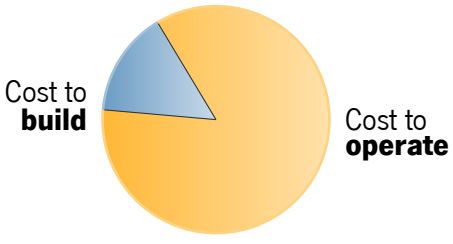


Figure 1.2: Cost Breakdown: Distribution of Building and Operating in Software Endeavors[4]

[4]Adapted from 'AWS re:Invent 2023 - Keynote with Dr. Werner Vogels': `https://maglit.me/cost`

Given that debugging is an ever-present and costly step in both software development and maintenance, automating it becomes imperative. Regarding fault localization, the methodologies often focus on 'where' issues occur, but the 'why' remains somewhat overlooked. Indeed, efforts to increase the granularity of components have been made, yet the contextual intricacies of code changes introduced with bugs are not fully explored.

After locating the source of a bug, we are still left with the task of fixing it. Automating this aspect has historically received less attention despite its critical role, as ensuring the generation of correct code is inherently more challenging than merely detecting bugs or locating their causes, requiring meticulous scrutiny and precision throughout preceding steps. In more recent years, APR techniques have been trying to tackle this issue. Still, they exhibit limitations in generalization, often being confined to specific bug types, languages, or programming constructs — a challenge akin to 'overfitting.' Thus, LLMs present themselves as a promising avenue. While LLMs excel at generating code akin to human writing, they are not inherently geared toward spontaneous repair. To harness these capabilities, we must intelligently explore code, extracting context to construct high-quality prompts for LLMs. The quality of these prompts directly influences the quality of the outputs produced.

With this in mind, the main question we want to address in this thesis is:

Main Question

How can we incorporate LLMs into APR by combining context from source code and FL?

To help us answer this question, we partition our efforts into four peripheral questions, each serving a distinct facet of the overarching investigation.

1.5.1 Understanding Failures

In modern software development, collaborative teams use sophisticated tools for productivity, introducing new features or updates through version control systems. Continuous integration tests result in a new build. Yet, unintentional bugs arise, leading developers to inspect source code. Identifying changes solely by analyzing modified lines, e.g. through diff reports, lacks contextual insight, thus hindering problem resolution. Upon deeper inspection, developers identify the kind of modification caused the bug. Frequently, these modifications unveil contextual insights and can be mapped to well-known transformations called mutation operators. These specific and well-

structured changes, like altering method invocations or control structures, offer valuable context, enhancing developers' understanding and facilitating solution development. Indeed, recent studies affirm the connection between real-world software faults and mutation operators (Just et al., 2014; Andrews et al., 2005; Daran and Thévenod-Fosse, 1996; Namin and Kakarla, 2011).

This sets the stage for the question:

Research Question 1

Can we describe a program's evolution using mutation operators? ***(Chapter 3)***

1.5.2 Contextual Repair

Inferring mutation operators is a crucial step in integrating APR into continuous integration systems, aiming to fix faulty programs by considering contextual modifications leading to bug introduction. Examining how contextual information shapes fault localization sets it apart from traditional SFL. Unlike SFL's focus on "where" faults occur, contextual insights explore the "why," providing a more comprehensive understanding of fault origins. This shift toward mutations and context potentially influences APR. Concentrating on mutations introduced during software evolution enriches APR with additional context behind faults.

With this in mind, we define the question:

Research Question 2

Does the information about inferred mutations benefit APR? ***(Chapter 3)***

1.5.3 Generative Repair

So far, our questions have revolved around the impact of inferring context on fault localization and program repair. However, a key focus of this thesis is the integration of LLMs into APR. To address this concern, we need to scrutinize the effectiveness of these AI tools in a more isolated manner. Specifically, we aim to explore how generated code can be utilized when fault localization is conducted flawlessly and relies solely on immediate and readily available context. Moreover, as we confront the challenges of repairing faulty programs, we encounter limitations in current techniques that prompt the exploration of innovative solutions. Patches are traditionally generated from developers' efforts that resulted in almost accurate programs. Stepping away

from traditional methods like evolutionary repair, constraints, and templates, known for their restrictiveness and potential limitations in producing effective bug-fixing code, we aim to explore a novel avenue. Thus, justifiably, our focus turns to LLMs, which are pre-trained on extensive code datasets. We aim to investigate the capability of these LLMs to generate patches that, when integrated into faulty programs, could prove effective in fixing faults. Although we anticipate a more flexible and creative solution to the inherent challenges of program repair, there is a need to evaluate the potential advantages.

As such, the following question arises:

Research Question 3

Can LLM-generated code evolve and fix faulty programs? ***(Chapter 4)***

1.5.4 Type Error Repair

Within programming languages, type systems play a crucial role in upholding correctness by ensuring the compatibility of operations with program terms. Despite their effectiveness in signaling logical errors through failed typechecking, type systems often fall short in precisely pinpointing and explaining type inconsistencies. The necessity for such exploration becomes apparent when considering that reported type inconsistencies may deviate from the user's intended modifications. Additionally, the left-to-right bias in error detection, influenced by the sequence of expressions in a program, introduces challenges that demand attention. Efforts have been done to enhance the quality of type error messages, highlighting the need for approaches to boost the accuracy and comprehensibility of type error resolutions. Even if we assume these limitations have been overcome, the task of rectifying such type errors remains. We wish to explore whether the code understanding capabilities of language models, augmented with information from type systems and their detected type errors, can effectively address and fix these issues. This marks a significant shift in focus, diverging from traditional APR approaches that concentrate on faulty programs capable of execution. Often overlooked are programs that fail to compile.

This is the basis for our last question:

Research Question 4

Can integrating type system information into LLM interaction fix type errors? ***(Chapter 5)***

1.6 Contributions

Overall, this thesis makes several contributions. In particular, we:

- Propose a mutation operator inference technique that examines changes between an original program and its updated bug-introducing version, making it easier to identify the responsible modifications. This technique is implemented as a tool, validated using a dataset and further complemented by a repair strategy and its corresponding tool. The significance of these contributions is highlighted through a case study investigation of real-world bugs, demonstrating the practical benefits of our approach in program analysis and repair. **(Chapter 3)**
- Redefine code completion, shifting its role from aiding developers in writing code to serving as a tool for APR. This involves using the code generation capabilities of pre-trained models to handle program repair as a code completion task. The tool implementing this repair technique fixes programs by determining the optimal places in a faulty line for code completion and seamlessly integrating the generated code. We test our effort on a real-world dataset and showcase its capabilities through case studies for which patches were automatically produced. **(Chapter 4)**
- Introduce an innovative approach that uses language models to automatically fix errors in OCaml programs, specifically the ones related to types. The methodology involves analyzing the program's source code and generating prompts with hints on how to create a corrected version. A publicly available tool supplements these efforts and enables their validation, with a discussion of the corresponding results made available. Unlike other methods that focus on improving error messages or letting people interactively fix mistakes, our approach uniquely automatically repairing these type errors. **(Chapter 5)**

1.6.1 Other Contributions

The contributions mentioned above, presented and explained in detail in their respective chapters, stem from the primary focus of this thesis. Apart from these, additional contributions were made throughout the course of this PhD, although they are not extensively discussed in this thesis. Nevertheless, they are noteworthy and deserve mention. Some directly align with the main thesis topic, while others diverge from it, yet remain pertinent.

Intrinsic Contributions

These contributions, though smaller in scale, are related to the overarching topic and their content is adequately addressed by the main contributions in the relevant chapters.

This contribution summarizes the efforts conducted in employing LLMs for APR and has been published, presented, and discussed at the Doctoral Symposium of SPLASH 2023. The contents are explored in detail in **Chapters 4 and 5**.

"Large Language Models for Automated Program Repair"
Francisco Ribeiro
In: Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2023
Cascais, Portugal, October 22–27, 2023
DOI: **10.1145/3618305.3623587**

This contribution is as a position paper advocating for the incorporation of type-awareness in LLMs, setting the stage for **Chapter 5**. Notably, as this thesis approached completion, significant works in prominent venues emerged (Zhu et al., 2023), echoing and implementing the same concerns. We believe this shows the relevance of the introduced ideas and suggests a shared recognition of these concerns within the members of the research community.

"Beyond Code Generation: The Need for Type-Aware Language Models"
Francisco Ribeiro, José Nuno Macedo, and Kanae Tsushima
In: 2023 IEEE/ACM International Workshop on Automated Program Repair, APR 2023
Melbourne, Australia, May 16, 2023
DOI: **10.1109/APR59189.2023.00011**

Separate Contributions

Some work does not directly align with the thesis topic, while remaining pertinent to many of the inherent objectives of this work.

The work "Ranking Programming Languages by Energy Efficiency" aligns with program repair concerns, as both prioritize efficiency. In certain contexts, optimizing for energy efficiency can be viewed as addressing a latent issue akin to fixing a bug in the code. This connection becomes

increasingly important, particularly in today's discussions around proper and adequate measures for sustainability, where one of the emphasis is on energy efficiency.

"Ranking Programming Languages by Energy Efficiency"
Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva
In: Science of Computer Programming, Volume 205, May 1, 2021, 102609
DOI: **10.1016/j.scico.2021.102609**

The integration of the functional paradigm in various scenarios in recent years has not only expanded its application but has also contributed to enhancing software sustainability. Under certain circumstances, where functions become a "first-class citizen," this shift towards modularization, in the form of functions or similar components, fosters better software maintenance practices. A prime illustration of this is the enhanced flexibility facilitated by property-based testing, a concept that commonly walks hand-in-hand with the functional paradigm.

"Functional Going Green: An Empirical Evaluation of Functional Languages Performance"
José Nuno Macedo, Francisco Ribeiro, Rui Rua, Marco Couto, Jácome Cunha, João Paulo Fernandes, João Saraiva, and Rui Pereira
In: Lectures Notes in Computer Science
(to be published)

1.7 Origin of Chapters

Apart from Chapters 1, 2 and 6, all chapters in this thesis have been published in peer-reviewed venues. All publications have been co-authored with João Saraiva and Rui Abreu. Publication of Chapter 5 has also been co-authored with José Nuno Macedo and Kanae Tsushima.

Chapter 3 has been published in the Proceedings of IEEE International Conference on Software Quality, Reliability and Security (QRS'21) (Ribeiro et al., 2021).

Chapter 4 has been published in the Proceedings of the International Workshop on Automated Program Repair (APR'22) (Ribeiro et al., 2022).

Chapter 5 has been published in the Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE'23) (Ribeiro et al., 2023a).

1.8 Thesis Outline

The first two chapters of this thesis are the foundation for the research work:

Chapter 1 Presents the motivation for this thesis, introduces relevant topics, states the main question it will target and the peripheral questions that will guide our research. Also, it lists the accomplished contributions and mentions their respective publications.

Chapter 2 Touches on the State of the Art directly pertaining to the areas this thesis aims to directly influence. Namely, fault localization and automated program repair.

The three contributions outlined in Section 1.6 are described in their respective chapters:

Chapter 3 Introduces a mutation inference technique, a complementary repair strategy and their corresponding validations.

Chapter 4 Showcases the use of a code-specific generative LLM to automatically fix programs by highlighting the necessary steps to generate and integrate the produced patches, alongside a validation.

Chapter 5 Presents a novel approach using prompt engineering to guide an LLM in fixing programs with type errors, validates the approach and compares it to relevant techniques.

In the end, the only thing left is to wrap up and conclude:

Chapter 6 Summarizes how the contributions answered the proposed main question and its more specific peripheral questions and discusses possible future lines of work.

State of the Art

2

2.1 Fault Localization

In software development, identifying and rectifying errors is a cornerstone activity. Central to this pursuit is the task of fault localization which can, most certainly, be regarded as the first step in the debugging process — one of the most expensive actions in the development cycle (Vessey, 1985). In this section, we will delve into the pivotal role that fault localization plays in software engineering. We will explore its significance, historical evolution and key aspects. Before moving further, note that effective fault localization not only expedites the debugging process but can also be used to inform subsequent stages, such as automated program repair. It acts as a guiding compass, directing APR techniques to the precise regions of code that require attention. This connection is of paramount importance as it underscores the symbiotic relationship between fault localization and automated program repair. By establishing a comprehensive understanding of fault localization, we pave the way for a more nuanced exploration of automated program repair, which we will delve into in the subsequent section.

2.1.1 Spectrum-based Fault Localization

In the realm of fault localization, one valuable and pioneering technique is *Spectrum-based Fault Localization* (SFL). Central to SFL is the concept of program spectra, first introduced by Reps et al. (1997). In essence, program spectra serve as a window into a program's execution behavior. These spectra are generated by a path profiler, which instruments a program to accumulate data on the unique loop-free paths that are executed during a program's runtime. In other words, it records what parts of a program are active during a particular run. Initially, the focus was on comparing program spectra between successful executions (where the program completed without errors) and unsuccessful executions (where errors were detected), all while keeping the same program but varying the data inputs, which ultimately determined whether an execution passed or failed. As a result, each program execution yields a distinct path spectrum, offering a snapshot

of the paths traversed during that specific run. These path spectra collectively provide a finite and precise characterization of a program's execution, encapsulating its behavioral signature for that run. One notable historical application of this approach was in addressing the Y2K problem, a famous bug in the field of software engineering. The Y2K issue arose due to the widespread practice of representing year values with only two digits in dates. This practice inadvertently led to ambiguous interpretations when processing year values such as '00', which could be interpreted as '1900' instead of '2000'. This phenomenon could result in faulty computations, especially in cases where calculations extended into the future. By comparing different execution spectra produced by different input dates (which passed or failed tests), software engineers could identify variations in program behavior related to date calculations. These divergences pointed them to specific sections of code where date-related issues might be lurking. While the Y2K issue was fundamentally a date-handling problem, the application of SFL showcased its utility in identifying and localizing software defects — a utility that has since become prevalent in modern fault localization practices.

While previous research primarily involved the comparison of hit spectra — whether a component is exercised or not — derived from executing the same program on varying input data, subsequent work (Harrold et al., 1998) expanded on the notion of hit spectra by introducing count spectra — number of times a component is exercised. Furthermore, the study goes beyond the usual path spectra consideration and introduces different kinds of program spectra, like branch spectra and execution-trace spectra. However, Harrold et al. (1998) use a different methodology to collect program spectra in which the original program and a slightly modified version of the same program are executed on identical input data. This means that attention now turns to regression testing, a scenario in which both the original, error-free version of the program and its subsequent, potentially buggy iteration are evaluated while maintaining the input data as a constant factor — a departure from prior work — which concentrated on different inputs to the same program. Enhancing granularity by utilizing execution trace spectra appears to yield more effective results in fault localization efforts. However, it's important to note that this type of spectra tends to exhibit greater variation, even in scenarios where no faults are present. While the increased sensitivity of execution trace spectra can be advantageous in detecting subtle defects, it also demands careful consideration to distinguish genuine anomalies from the natural variability that can arise during program execution. Striking the right balance between granularity and

specificity is a crucial aspect of precise fault localization, especially under real-world market conditions where exhaustive testing may be impractical. In such scenarios, software cannot always undergo exhaustive testing and high-impact bugs require prioritization.

Generally speaking, when disparate spectra emerge from different executions, it suggests divergences in program behavior. These divergences may not inherently pinpoint the root cause of a defect, but they serve as critical indicators for the initial point of interest for further investigation by programmers towards the regions where anomalies may reside. Thus, the power of SFL lies in its ability to compare different execution spectra.

An indispensable component of this comparative analysis task revolves around the calculation of similarity measures between various vectors within the program spectra data and a vector containing vital information about detected errors. Comparing and contrasting different spectra is accomplished through the application of similarity coefficients. These coefficients serve as quantitative metrics that assess the likeness or resemblance between vectors representing different aspects of program execution behavior. Abreu et al. (2006) study and report the accuracy of various similarity coefficients when applied to block hit spectra — a type of program spectra that records whether a block is executed or not. This investigation considered a range of similarity coefficients, including Jaccard (Chen et al., 2002), Tarantula (Jones et al., 2002), AMPLE (Dallmeier et al., 2005), and Ochiai (Meyer et al., 2004). It's worth mentioning that Ochiai, while not traditionally associated with the field of software engineering and borrowed from molecular biology, outperformed other coefficients in this context. On average, it exhibited a 5% improvement in correctly locating anomalies, with specific instances showcasing a 30% enhancement.

2.1.2 Model-based Fault Localization

The concept of model-based fault localization (MBFL) (de Kleer and Williams, 1987; Wotawa et al., 2002) represents a diagnostic approach where differences between a model of a system and the actual observed behavior of that system guide the search for discrepancies. This approach requires a comprehensive model of the system's structure and function. While it can provide superior diagnostic accuracy compared to SFL (Abreu et al., 2009), modeling complex systems can be exceptionally intricate and, many times, simply infeasible in practice. Consequently, many software diagnosis techniques tend to be black box approaches, which do not rely on in-

depth knowledge of the system's internal structure and are easily integrated with existing testing methods.

In contrast, white box techniques, such as MBFL, require understanding the internal components and behavior of the system. While they hold the potential for more accurate diagnosis, the complexity and cost associated with modeling can pose significant challenges in practical software development scenarios.

The idea of combining model-based software debugging (MBSD) with dynamic program analysis techniques, as discussed when presenting SFL, has been explored (Mayer et al., 2008). MBSD can guide automated debugging efforts but may generate large result sets, making it challenging to rank and discriminate between potential explanations for faults. While promising, integrating MBSD and dynamic program analysis still faces the inherent complexity of modeling. Indeed, formal specifications for non-trivial systems that accurately capture the semantics of said system can be as hard as implementing the system in question (Musuvathi and Engler, 2003).

In a more refined hybrid approach, a framework known as DEPUTO (Abreu et al., 2009) combines SFL with MBFL. Here, SFL is used initially to rank likely faulty components, inheriting its low computational complexity. Subsequently, MBFL is employed to refine this ranking by eliminating components that do not explain observed failures, inheriting the inherently enhanced diagnostic in MBSD. This integrated approach aims to strike a balance between computational efficiency and diagnostic precision, leveraging the strengths of both techniques to enhance fault localization.

2.1.3 Mutation-based Fault Localization

Although SFL is often regarded as lightweight and adaptable, making it easy to integrate with intended systems due to its black box nature, its limitations (Parnin and Orso, 2011; Ang et al., 2017) are diverse and have prompted a search for more accurate and efficient fault localization techniques. These limitations include:

- **Diagnostic Inaccuracy:** Despite extensive community efforts, SFL is still perceived as a relatively inaccurate method for pinpointing software faults. It relies on statistical analysis of program execution traces, which can introduce imprecision when attempting to identify the exact location of a fault within the codebase. This imprecision can lead to false positives

or false negatives in fault localization results.

- **Test Case Dependency:** One of the prominent challenges with SFL is its heavy reliance on the availability of comprehensive and diverse test cases. To effectively employ SFL, a sufficient number of test cases covering various program behaviors and edge cases are required. Unfortunately, in many real-world scenarios, obtaining an exhaustive set of test cases can be a daunting task. The limited availability of test cases can undermine the accuracy and reliability of SFL.
- **Long Lists of Candidates:** SFL often generates ordered but lengthy lists of statements, especially when faults are complex or deeply embedded within the codebase. These lists can be overwhelming for developers tasked with debugging and can include a substantial number of statements unrelated to the actual fault. This inundation of information can lead to a phenomenon where developer interest drops over time.
- **Diminished Developer Engagement:** The extensive and sometimes convoluted lists produced by SFL can lead to a drop-off in developer engagement. Developers may lose motivation and interest in the fault localization process when they perceive it as a time-consuming and unproductive endeavor, causing them to abandon this process prematurely.

In response to these limitations, researchers have explored alternative fault localization approaches, such as mutation-based fault localization, which aims to provide more precise and actionable fault localization results. These alternative approaches seek to address the challenges associated with SFL and offer improved accuracy, reduced dependency on test cases, and more comprehensible fault localization reports to enhance developer engagement and software debugging efficiency.

Program mutation

Before understanding mutation-based fault localization, we must first get into the definition of program mutation.

Program mutation is a software analysis technique pioneered by DeMillo et al. (1978) and initially introduced within the context of software testing. In essence, it revolves around the creation of variants, known as mutants, from a target program. These mutants are generated by applying specific, predefined syntactic changes, or mutations, to the original program's code.

The primary objective of program mutation is to assess whether a given test suite adequately tests a program. In simpler terms, it's a way to "test the tests", and is a way to build confidence on the quality of a test suite. More broadly, it confronts the more general philosophical question of "Who will guard the guards?". The process involves two possible outcomes for each mutant and assumes the original program passes the provided test suite:

- If a mutant produces a different result from the original (target) program when subjected to the test suite, it is deemed **dead**. This signifies that the test data successfully distinguished the mutant from the original program.
- Conversely, if a mutant and the original program produce the same result under the test suite, the mutant is **alive**. This can suggest that either the existing tests lack the sensitivity to detect the error introduced by the mutant, or the mutant and the original program are equivalent and, therefore, indistinguishable in this scenario.

Ultimately, a test suite is considered adequate if it can "kill" every mutant, meaning that all mutants introduced are detected by the tests. Alternatively, the test suite may let some mutants survive and still be deemed adequate if those mutants are considered equivalent to the original program.

While the definition of program mutation is closely associated with software testing due to its origins, it has found applications in various domains and is sometimes referred to as mutation analysis. In a broader sense, mutation analysis entails the application of well-defined transformations to the syntactic structure of a program, resulting in variants that exhibit minor semantic differences from the original program. This general definition highlights the versatility of mutation analysis beyond its initial testing-centric context, making it a valuable tool in diverse software engineering and analysis scenarios, namely fault localization.

Mutating Faulty Code for Fault Localization

Mutation-based fault localization presents a distinct approach compared to traditional mutation testing. In mutation-based fault localization, the process revolves around applying mutations to programs that are already known to be faulty, with the goal of pinpointing the specific faults. This differs from mutation testing, where mutations are injected into an ostensibly correct program to evaluate the effectiveness of the test suite.

If we think about the basic concept of fault localization in simple terms, it mainly boils down to one idea: segments of code covered by failing tests are more likely to harbor faults than segments predominantly covered by passing tests. Mutation-based fault localization builds upon this foundation. The fundamental concept is that by generating variant programs from a known faulty one through the mutation of specific statements, it becomes possible to identify the fault's location. When faulty statements are mutated, more failing tests are expected to pass, whereas mutating correct statements (i.e., where the fault is not located) is anticipated to cause previously passing tests to fail.

Existing techniques approach this in different ways. In a tool like Metallaxis (Papadakis and Le Traon, 2012, 2015), mutations are applied to various statements within a faulty program. The test suite is executed for each mutated program, and suspiciousness values are assigned to individual statements employing the Ochiai coefficient. However, there is a challenge in assigning suspiciousness values to the statements of the original (faulty) program. Each original statement may have multiple associated mutant statements, with this number depending on the applicable mutation operators for a given original statement. In such cases, Metallaxis assigns the highest suspiciousness value among all the associated mutants to each original statement.

On the other hand, MUSE (Moon et al., 2014) adopts a different approach. Its focus lies in analyzing the outcomes of individual test cases. MUSE examines scenarios where mutants transform initially failing test cases into passing ones, and vice versa. The underlying principle is that a faulty program can be rectified by modifying (mutating) the problematic statements. Therefore, the intuition is that applying mutations to faulty statements should cause more failing test cases to pass. Conversely, mutating correct statements is more likely to induce previously passing test cases to fail, effectively injecting new faults. In generating its ranking of suspicious statements, MUSE disregards test case results and focuses exclusively on the impact of mutations on test case outcomes.

Metallaxis and MUSE primarily focus on the C programming language. Indeed, modern software projects often involve multiple programming languages. This multilingual approach allows developers to harness various control and data abstractions, as well as utilize legacy libraries. The software landscape has evolved considerably in recent decades, and complex programs are frequently composed of code written in multiple languages. Hong et al. (2015) have applied mutation-based fault localization to such multilingual environments. Their approach includes the

development and application of both new and conventional mutation operators, with experiments conducted on six real-world multilingual programs showing that bugs can be located accurately for such subject programs. In fact, some approaches (Hanam et al., 2016) mine project repositories to find frequent bug patterns for languages that research has yet to report mutation operators.

In a different approach, Zhang et al. (2013) introduce a technique that involves the mapping of artificially generated mutants to higher-level programming edits. The fundamental idea behind this approach is to capture the core characteristics of failure-inducing edits made by developers using mechanical program transformations, such as mutation changes. The authors address this challenge within the context of evolving codebases, where code is continually updated. Their solution leverages the notion that certain transformations are likely to coincide with failure-inducing edits if these transformations alter the old program version (i.e., the version just prior to the introduction of faults) to produce similar test pass/fail results as the new version with actual developer edits. Through experiments conducted on nine real-world Java projects, the authors demonstrate that strategies combining spectrum information and injected faults outperform the previous approach that relied solely on spectrum data.

Mutants as real faults

Real-world programs with actual faults can be challenging to acquire. While such programs do exist, they may not always be readily available for experimentation. Even when real programs and associated faults are accessible, they may not be available in sufficient quantity. This scarcity of faults can hinder the conduct of experiments, as small-scale experiments may lack the statistical significance required for robust conclusions.

Due to these limitations, it is common practice to introduce faults deliberately into the programs under investigation. Faults can be introduced manually or automatically. Manual fault introduction, although potentially more realistic, comes with several drawbacks. First, the realism of the introduced faults can be subjective, and different individuals may hold varying opinions about the authenticity of the fault. Second, it is challenging to reproduce experiments conducted by other researchers, limiting the reliability of experimental science. Lastly, manual fault introduction can be time-consuming and impractical for generating a substantial number of faults.

Program mutation offers a solution to some of these challenges. It employs well-defined

operators to precisely seed faults into programs, enhancing the ease of replicating experiments. Additionally, program mutation enables the generation of a large number of mutants relatively easily, improving statistical significance in experiments.

However, a pivotal question remains: Are artificial mutations representative of real-world faults? The answer to this question holds significant importance. If artificial mutations do not accurately emulate real-world changes, the conclusions drawn from experiments employing program mutation may not be valid. As such, the authenticity and realism of mutations as fault representations play a crucial role in the credibility of experimental findings in software analysis.

Some studies have been conducted to determine the suitability of mutants as proxies for genuine faults.

Initially, this question was not addressed (Daran and Thévenod-Fosse, 1996) in terms of whether mutations could serve as a representative fault model for real faults. The prevailing perspective seemed to lean towards a negative response to this inquiry. Instead, the primary emphasis was placed on ascertaining whether errors and failures induced by mutations exhibited similarities to those associated with real faults. The results support the representativeness of errors generated by mutations, with 85% of these errors corresponding to those produced by real faults, highlighting the ability of mutations to trigger error behaviors of comparable complexity to real faults. In an earlier work by Offutt (1989), empirical investigations into the coupling effect were conducted, providing evidence for it. Research concluded that test data generated to uncover simple faults, such as first-order mutations, could effectively reveal more complex mutations. Similarly, DeMillo and Mathur (1990) had already recognized that mutations had the potential to induce significant variations in a program's internal state during testing executions. These variations played a pivotal role in uncovering complex faults within the software.

In another study (Andrews et al., 2005) aimed at evaluating the appropriateness of mutation analysis for software testing experiments, researchers compared test suites' fault detection capabilities across three distinct fault categories: hand-seeded faults, automatically generated mutants, and real-world faults. To conduct this assessment, the researchers used a set of widely-used software programs featuring hand-seeded faults, as well as a widely-recognized program known to exhibit real-world faults. The findings demonstrated that the generated mutants were representative of real-world faults, setting them apart from the hand-seeded faults. Moreover, an interesting aspect was also shown: hand-seeded faults proved to be considerably

more challenging to detect when compared to their real-world counterparts. Namin and Kakarla (2011) addressed these findings and the controlled experiments conducted in their paper yield complementary insights, demonstrating the high sensitivity of mutation in testing experiments to external factors, including mutation operators, test suite size, and programming languages. These findings highlight the importance of considering these factors when interpreting or generalizing experimental results based on mutation analysis. The study deliberately employed the same subject programs as (Andrews et al., 2005) to minimize potential bias arising from the use of different artifacts. Furthermore, the utilization of a distinct mutation tool, featuring a greater number and more complex mutation operators enriches the experimental context.

On a related note, another one of these studies (Andrews et al., 2006) centered on an industrial software program with documented faults. The principal aim was to assess whether mutation analysis could viably serve as a means to measure the cost-effectiveness of control and data flow criteria in software testing. The findings suggest that mutation analysis exhibits considerable potential for this. Most notably, the study revealed that mutation analysis often yields results analogous to those derived from actual faults. This underscores the value of mutation analysis in a research context, enabling the comparison and evaluation of emerging testing techniques. Moreover, it also helps determine the necessary coverage levels to attain acceptable fault detection rates.

One consequential finding from this investigation was the absence of a single control or data flow criterion that demonstrated unequivocal superiority in terms of cost-effectiveness. Nevertheless, the study discerned a somewhat expected finding: more intricate criteria, necessitating larger test suites, were more effective at detecting a greater number of faults.

Another related work (Just et al., 2014) focused on the correlation between a test suite's mutation score and its real fault detection rate. A general correlation between a test suite's mutation score and its proficiency in detecting genuine faults was observed. Notably, this correlation was found in a substantial portion of the real faults under examination, approximately 73%. Furthermore, the research emphasized particular mutation operators — such as conditional operator replacement, relational operator replacement, and statement deletion — which displayed a more frequent association with genuine faults. Drawing from these insights, the researchers provided suggestions for potential enhancements to mutation analysis. They proposed an in-depth exploration of uncoupled faults and advocated for the inclusion of class-level mutation operators (Offutt

et al., 2006; Ma et al., 2002; Kim et al., 2000) to augment the efficacy of mutation analysis.

It is worth noting, however, that class-level mutation operators, while valuable, are not as uniformly implemented in mutation tools nor used in experiments as their traditional counterparts. The authors argue that including them could hinder the ability to compare and apply findings because class-level operators, which are more specific, are not as widely applicable across different programming languages as the broader set of traditional operators.

Higher-order mutants

Jia and Harman (2009) introduced the concept of higher-order mutation testing, a novel approach in which mutants are not individual faults but instead comprise multiple faults. This approach aims to identify those intricate higher-order mutants that represent subtle and challenging to detect faults. Achieving this involves employing automated search-based optimization techniques to identify combinations of simple faults that partially mask each other, making them collectively more challenging to detect than their individual constituent faults. Such fault combinations are relatively rare but hold significant value in software testing.

In this scenario, mutants are classified into two categories: First Order Mutants (FOMs), generated by applying mutation operators only once, and Higher Order Mutants (HOMs), generated by applying mutation operators multiple times. Within this context, the authors focus on a specific type of HOM known as subsuming higher-order mutants, which are notably more challenging to eliminate than the FOMs that compose them.

Subsuming HOMs can only be eliminated by a subset of the test cases that detect each individual FOM from which they are constructed. There is also the concept of strongly subsuming HOMs, where a single HOM can replace all of the mutants from which it is derived without compromising test effectiveness. This concept highlights the efficiency and effectiveness of higher-order mutation testing.

Higher-order mutants offer multiple advantages in software testing. First, they enhance fault detection by introducing increased subtlety into the testing process. While most First Order Mutants (FOMs) are easily identified by simple test cases due to their often trivial nature, subsuming HOMs represent more intricate faults. These subtle faults may remain concealed during standard testing but become apparent when exploring challenging corner cases that are more likely to be

missed.

Second, higher-order mutation testing reduces the overall test effort required. Although there are exponentially more HOMs than FOMs, this potential increase in testing complexity is effectively mitigated by targeting strongly subsuming HOMs. These powerful mutants can replace multiple FOMs, resulting in fewer mutants to consider without sacrificing the quality of test cases.

Third, higher-order mutants lead to a reduced number of equivalent mutants, which are mutants that cannot be distinguished from the original program by any test input. While identifying equivalent mutants is generally challenging, second-order mutation testing has demonstrated a significantly lower density of equivalent mutants compared to first-order mutation testing (Offutt, 1992). This reduction, often overlooked in prior research, contributes to a substantial reduction in test efforts by up to 50% while maintaining effective fault detection (Polo et al., 2009).

Program Mutation for Bug Detection

The role of test cases in modern software development cannot be overstated, as they serve as a fundamental tool for detecting software defects. While the primary focus here is on bug detection rather than localization, it's worth noting the relevance of program mutation in the context of test case generation. After all, effective fault localization is only possible when bugs are initially detected. This underscores the versatility of program mutation as a technique that contributes to various aspects of software quality.

As discussed earlier, program mutation is frequently employed to assess the quality of test cases. When mutation analysis reveals that certain non-equivalent mutants remain unaddressed, it signals the need for test case improvement. This process demands an in-depth understanding of the source code and is a non-trivial task. While automated test generation can assist in code coverage and mutant detection, the tester must still evaluate the outcomes of these generated executions.

Fraser and Zeller (2010) introduce an approach that automates the generation of unit tests for object-oriented classes by leveraging mutation analysis. Their strategy uses mutations, rather than structural properties, as a coverage criterion. This not only guides testers on where to test but also what specific aspects to test for.

Furthermore, the application of mutation-based test case generation extends beyond tradi-

tional software development. It has found utility in domains such as web page testing (Almeida et al., 2019). In fact, tools like Sapienz (Mao et al., 2016) are already implementing this approach successfully, identifying bugs in widely-used mobile applications that impact millions of users.

2.1.4 Qualitative Spectrum-based Fault Localization

Recall that SFL is a methodology that associates a system's components with observed failures. This approach offers a lightweight means of identifying potential faults by analyzing code coverage.

Despite the advancements and accomplishments in SFL research, its widespread adoption in the industry has been limited. This limitation is primarily driven by several issues identified in prior studies (Parnin and Orso, 2011; Ang et al., 2017), such as the substantial decrease in user interest after examining a small portion of the ranked list of potential faults. This problem becomes more pronounced as the complexity of the system grows. Another challenge is that many SFL studies assume perfect fault comprehension, expecting that users will correctly identify faulty components upon inspection. However, this assumption is not always verified in practice.

In fact, the abstractions that SFL incurs in also present other limitations. It may overlook certain types of faults, such as omission errors, and often falls short of providing sufficient context to explain why specific components are deemed suspicious.

Thus, it is imperative to augment fault localization approaches with more contextual information, so that we can increase fault comprehension.

Q-SFL (Perez and Abreu, 2018), an enhanced approach building upon SFL, takes a qualitative perspective on system components. In this methodology, system components are qualitatively partitioned, treating each qualitative state as a distinct SFL component for diagnostic purposes. The driving force behind this approach is Qualitative Reasoning (QR) (Forbus, 2004; Williams and de Kleer, 1991; de Kleer, 1990), a technique for describing sets of values based on their discrete behavioral qualities. Unlike traditional numerical quantities, which can be unmanageable to record for each individual component in a system, QR employs qualitative descriptions like "high," "low," "zero," "increasing," or "decreasing."

These qualitative descriptions, also referred to as quantitative landmarks, partition the domains of relevant components into distinct descriptions, each forming a new SFL component. As behavioral qualities are now considered as components themselves, their involvement in system

executions is recorded and ranked based on their similarity to observed failures. This enrichment of the SFL report aligns with the initial goal of providing more comprehensive diagnostic information to aid in fault comprehension.

This approach holds significant implications for fault comprehension because it not only records the involvement of qualitative properties but also considers their correlation with failing behavior. When a qualitative state exhibits a stronger correlation with failing behavior than its enclosing system component, this information can be extracted.

Empirical evaluation of Q-SFL has demonstrated its potential to enhance the accuracy of SFL reports. In this evaluation, 54% of the considered subjects exhibited a reduced effort in diagnosing faults. The study utilized program subjects from the Defects4J dataset, executing fault-revealing test suites for each buggy subject and constructing method-coverage spectrums that, naturally, encompass test outcomes. Additionally, the values of primitive-type arguments and return values for every method call were recorded to support the creation of various landmarks.

Several partitioning strategies were explored, resulting in multiple variants of Q-SFL. These strategies included static partitioning, which employed automated sign partitioning based on variable types, and dynamic partitioning, using various clustering and classification algorithms such as k-NN, linear classification, logistic regression, decision trees, random forest, and x-means clustering.

2.2 Automated Program Repair

As discussed in Section 1.3 of the Introductory chapter, automated program repair plays a pivotal role in addressing the challenges of modern software development (Goues et al., 2019). In this chapter, we will take a closer look at the foundations of automated program repair. While the introduction offered an overview of this field, this chapter aims to provide a more in-depth understanding. We will explore the historical developments, foundational concepts, and key techniques that underpin automated program repair. This detailed background is essential for a comprehensive grasp of the subject and will serve as a solid foundation for the subsequent chapters of this thesis. In the following sections, we will delve into the intricacies of automated program repair by exploring previous important works, ultimately setting the stage for our research contributions.

2.2.1 Evolutionary Repair

Genprog (Le Goues et al., 2012b) stands as an influential and characteristic automated program repair system that harnesses the power of genetic programming (GP) to evolve program variants with the objective of rectifying known defects while preserving essential functionality.

At its core, Genprog takes as input a program, a set of positive test cases that exemplify the desired program behavior, and a negative test case that detects the fault. GP, inspired by biological evolution, operates by maintaining a population of individual programs. These programs undergo variations akin to biological mutation and crossover, which yield program variants. The fitness of each program variant is evaluated through a user-defined fitness function, ensuring that only the most promising variants progress in the evolutionary process. GP has been demonstrated to successfully address a diverse array of computational challenges (Hum, 2004).

In the context of Genprog, GP is employed to curate a population of program variants intended to rectify the target program. Each variant is represented as an abstract syntax tree (AST) paired with a weighted program path. Two specific genetic algorithm operations, crossover and mutation, tailored to this representation are employed to modify program variants. Each modification results in a new AST and weighted program path. The fitness of these variants is assessed by compiling the AST and executing the program on the provided test cases. The final fitness score is computed as a weighted sum of the positive and negative test cases that the variant successfully passes. The evolutionary process continues until a program variant is produced that satisfies all test cases, effectively resolving the defect.

Genprog introduces two fundamental considerations. First, it constrains the evolutionary algorithm to generate changes that are rooted in structures found elsewhere in the program. The guiding hypothesis is that a program missing essential functionality can often borrow and adapt the necessary code from other sections of the program. This premise arises from the observation that a program may correctly handle certain situations in one location even if it makes an error in another (Engler et al., 2001). For instance, if a program lacks a null check or an array bounds check, it may have a similar working check elsewhere that can serve as a template. Consequently, only code similar in structure to existing code is inserted. Thus, specific if conditionals and precise statements may be inserted given that these are present in the original program. Second, Genprog restricts the genetic operations, mutation, and crossover, to operate exclusively on the part of the

program that is relevant to the error, which are computed by the execution path leading to the error. In other words, when selecting sections of the program to modify, those visited during the negative test case but not visited during the positive cases are favored. These two critical factors attempt to overcome a substantial challenge inherent in evolutionary algorithms like GP, which is the need to traverse a potentially boundless search space in pursuit of a correct program.

To generate candidate program variants, Genprog operates on the program's AST and applies three types of operations: deletion, addition, and replacement of AST nodes. When adding or replacing nodes, they are sourced from other parts of the codebase, aligning with the redundancy assumption (Barr et al., 2014; Martinez et al., 2014), which ensures that existing, functional code is leveraged in the repair process.

Genprog's initial evaluation involved 10 C programs from open-source benchmarks, all of which were reported to have been successfully fixed. Later, a more extensive evaluation was conducted, covering 105 defects across 8 C programs, with 55 defects being automatically repaired according to the authors (Le Goues et al., 2012a).

However, despite its significance in the APR field, subsequent studies have scrutinized the patches generated by Genprog and discovered a high rate of incorrect patches (Qi et al., 2015). This phenomenon raises an open question about whether this holds true for test-suite-based repair methods in general (Martinez et al., 2017).

In cases where patches initially reported as successful repairs are found to be incorrect, it is typically due to their reliance on specificities and vulnerabilities within the test suite, akin to overfitting. Overfitting occurs when a synthesized patch functions only on failing inputs and fails to generalize to broader cases. Furthermore, research by Smith et al. (2015) focused on the overfitting issue in automatic repair, demonstrating that Genprog and related techniques are indeed susceptible to overfitting, as they illustrated on a dataset of student programs.

The challenge of overfitting stems from the inherent limitation of techniques that focus on the immediate program context they are attempting to fix, which can impede the evolution towards meaningful solutions. Moreover, the process of mutation in Genprog may frequently produce nonsensical patches (Kim et al., 2013).

Jaff (Arcuri, 2011), a research prototype, takes a similar but still somewhat distinct approach. It models the repair task as a search problem and employs evolutionary algorithms to tackle it.

Jaff's focus centers on a subset of Java, using different search algorithms for comparison.

Three distinct search algorithms are considered and compared. These include a random search serving as a baseline, a variant of Hill Climbing, and Genetic Programming. Similar to Genprog, Jaff also strives to concentrate repair efforts around potentially problematic areas within a program. To facilitate this, the authors introduce a novel search operator based on fault localization techniques. This operator plays a crucial role in narrowing down the search effort to promising sub-areas of the search space.

The authors are well-aware of the challenges posed by the vastness of the search space and the risks of overfitting, where a program becomes too tailored to its initial test cases. They acknowledge that using conventional GP in its standard manner may be insufficient to evolve a correct program from scratch, especially given a set of test cases. However, the assumption that programmers do not create programs at random (DeMillo et al., 1978) makes the task less daunting. With this in mind, it is reasonable to assume that most repair sequences are not excessively long, reducing the complexity. Starting with a solution close to a global optimum influences the choice of search operations. In many GP applications, crossover is favored over mutation, but in the case of Jaff, the scarcity of diversity in the population calls for a different strategy. Hence, the focus is given to mutations instead.

Jaff works with program trees, specifically defining six mutation operators to modify abstract syntax trees. During a mutation event, one of these six operators is chosen with uniform probability. This approach includes an initial investigation and outlines directions for future work. To provide an initial validation, the authors evaluate their approach on toy programs and present a case study. Notably, it's highlighted that Genprog would be unable to repair the specific case study, primarily because Genprog's operators are ill-suited to address the faults in question used in that case study.

Tackling Overfitting through Property-based Testing A common way to assess whether a candidate patch is a valid fix is to execute a test suite (if available). However, there may be the case where a patch passes all tests but does not provide all intended functionality. We already discussed what overfitting is and described it as a scenario in which a patch, despite behaving as expected for the provided input data, is so specifically implemented that it fails to generalize.

PropR (Gissurarson et al., 2022) is a technique that targets the overfitting problem by using

property-based testing (Claessen and Hughes, 2000).

In property-based testing, instead of writing specific test cases, we define properties for a program. A property is a specific behavior or characteristic that a program is expected to exhibit. A property for the `reverse` function, which inverts the order of the elements in a list of integers, can be defined in Haskell as:

```
1 prop_reverse :: [Int] -> Bool
2 prop_reverse xs = xs == reverse (reverse xs)
```

The property `prop_reverse` states that for any list of integers, reversing the list twice should result in the original list, ensuring the correctness of the `reverse` function. To do that, multiple lists of varying lengths are automatically generated to assess the function's behavior. Generally speaking, based on the defined properties, input data is randomly generated, which is used to check if such properties hold true. Unlike traditional testing, which focuses on specific input-output pairs, property-based testing explores a broader set of inputs to ensure that certain properties remain valid. In theory, property-based testing can be applied to programs written in virtually any programming language. Indeed, there are frameworks for a diverse array of programming languages that support this testing approach. Still, it is a concept that is notably more common in the functional programming paradigm. A well-known and established framework for property-based testing in Haskell is QuickCheck with several adaptations for different languages.

For a given program to be repaired, PropR traverses its AST in order to determine the tests and properties to analyze. To determine the function bindings to mutate, we traverse the ASTs of the properties and find variables that refer to top-level bindings. These bindings will be the targets for repair. Consider the following faulty implementation of `len` and two corresponding properties:

```
1 len :: [a] -> Int
2 len [] = 0
3 len xs = product $ map (const (1 :: Int)) xs
4
5 prop_abc :: Bool
6 prop_abc = len "abc" == 3
7
8 prop_dup :: [a] -> Bool
9 prop_dup x = len (x ++ x) == 2 * len x
```

This implementation is incorrect because it calculates the length of a list by multiplying the

elements' contributions (`product`), instead of summing them (`sum`). Both properties refer to `len`, so it will be the target in this case.

Fault localization is performed at an expression-level. Specifically, expressions involved in the execution of a failing property are considered suspicious, to which the authors call fault-involved expressions.

The next step is called perforation. For each expression computed in the previous fault localization step, a variation of that program's AST is produced in which only that expression is replaced by a typed hole. A typed hole is a placeholder in Haskell code — represented by an underscore (_) — that the compiler recognizes as incomplete. During compilation, we can obtain type-related information about the typed holes. Moreover, we can also get valid-hole fits which are potential completions for typed holes that adhere to the expected types in the code and take into account the programming context in question, that is, the scope. Considering a typed hole is introduced for the `product` function:

```
1 len xs = _ $ map (const (1 :: Int)) xs
```

`ghc` will tell us that one of the valid-hole fits is `sum`. This would be computed by using depth 1. PropR supports using greater values for depth, meaning we could further introduce typed holes in hole-fits, which would allow to recursively refine hole-fits with more elaborate expressions.

A limitation is that the majority of specific constants and expressions requiring intricate expansions are not included in these valid-hole fits. To overcome this, PropR also generates expression candidates for certain types such as common values like 0, 1, and list concatenation ++ and application candidates according to the `_ x` template in which `x` is an already existing expression. After finding candidate fixes, the corresponding typed holes are replaced, thus building new targets. For the example used so far, replacing the typed hole with the valid-hole fit `sum` would yield the target:

```
1 len xs = sum $ map (const (1 :: Int)) xs
```

Interestingly, even though PropR has a strong focus on types, the generated patches may not necessarily compile. According to the authors, one of the causes may be an error in precedence, a task performed after typechecking. If a target passes all properties, it qualifies as a possible repair.

PropR implements three search algorithms. Random search randomly selects evaluated holes and valid matches for these holes are randomly chosen to be filled within them. Exhaustive search systematically examines each hole-fit in a breadth-first manner, starting with single replacement fixes, followed by two replacement fixes, and continuing until the search budget is depleted. Genetic Search involves mutations such as dropping or adding replacements to a fix. The fittest candidates, comprising the top $x\%$, progress to the next generation, with the remaining $(100x)\%$ filled by random individuals. High mutation and crossover rates are used to introduce substantial changes, as altering a single expression often causes more failures than fixes in properties.

PropR was evaluated on a dataset with 30 programs written by students for a programming task implementing a calculator that parses a term from text, calculates results and derivations. Even though it is a small scale classroom exercise, the authors argue that the problem encompasses sub-problems (like parsing) are complex enough to be representative of real-world scenarios and worthwhile to explore. The programs in question fail at least one of 23 properties and one of 20 unit tests. PropR was able to find patches for 13 buggy programs, producing 228 patches in total. In the study, it was observed that properties did not contribute significantly to patch generation compared to unit tests. Unique patches were identified by properties that were not generated by unit tests, and the disparity in results between genetic and exhaustive searches was most pronounced for properties. Genetic search demonstrates faster patch discovery for properties compared to unit tests, and the combined test suite further enhances the overall search speed. 70 patches were sampled, with 49 being identified as overfit and 21 as fit. Considering the overall population of 228 patches and a 10% error rate, the expected correct patches range from 62 to 76, resulting in a total non-overfit rate of 27% to 33%. Specifically, unit test-found patches were overfit in 85% of cases, while properties showed a lower overfit rate of 64%. The distribution varies across programs, with some consistently well-repaired and others prone to overfitting. Deductively, of the 13 entries with fixes, an estimated 3 to 4 have non-overfit repairs, yielding an effective repair rate of 10% to 13%, comparable to Astor's rates and superior to GenProg. The addition of properties and combining with them reduced the overfit ratio from 85% to 63%, leading to a twofold increase in the number of successful patches. Despite overfitting, the resulting repair rate of 10% to 13% with properties is comparable to other tools, indicating the overall strength of the test suite.

Overall, PropR presents an interesting exploration with promising results and a robust proof of concept. Still, the limited validation, with the dataset originating from the same team that developed the approach, and the percentages derived from sampling and manual inspection being generalized as effective rates for the method being presented raise some questions, emphasizing the need for a more diverse and rigorous evaluation.

2.2.2 Template-based Repair

PAR (Pattern-based Automatic Program Repair) offers an automated approach for the rectification of Java code defects. In APR, the degree of randomness associated with evolutionary techniques often leads to the generation of nonsensical patches. This inherent limitation highlights the necessity for evolved patches to exhibit qualities that go beyond mere satisfaction of test suites.

Effective patches should encompass a more profound understanding of the problem at hand, ensuring that the modifications generalize the effect of the patch. Additionally, patches should avoid overfitting to specific erroneous inputs and maintain clarity for human developers to understand and approve.

PAR operates based on a set of repair templates, each representing a common method for addressing prevalent types of bugs. For instance, one of these templates, known as the "Null Pointer Checker," is frequently applied to resolve null pointer access issues by inserting a nullness check before the problematic access. These templates can also be parameterized, accommodating variable names as inputs, for greater flexibility in addressing various issues.

Much like other automated repair approaches, PAR initiates its process by identifying fault locations, or suspicious statements, using existing fault localization techniques. The adjacent code surrounding these locations is then subjected to modifications, employing fix templates to create program variants, which are essentially patch candidates. These candidates are then evaluated using a fitness function that computes the number of passing test cases for each one. A patch candidate successfully passes if it satisfies all provided test cases; otherwise, the process of generating candidates and evaluating them is iteratively repeated.

To validate its effectiveness, PAR was empirically evaluated on a set of 119 real-world bugs, successfully fixing 27 of them. For comparison, Genprog, the tool the authors used as a baseline, managed to repair 16 of the same bugs.

2.2.3 Constraint-based Repair

Semfix (Nguyen et al., 2013) is a program repair approach in which a test suite is available to verify program correctness. In fact, the absence of a formal specification is generally the most common scenario.

The repair process in Semfix is guided by the generation of repair constraints, with the primary objective of ensuring the produced program successfully passes the given test cases. Following the pattern of other APR approaches, Semfix initiates its workflow with a fault isolation step, identifying potential locations for the problem. The suspicious lines of code are computed by leveraging the ranking produced by a statistical fault isolation tool.

The subsequent step involves the automatic discovery of the correct specification for the faulty statement using angelic debugging (Chandra et al., 2011), sometimes called predicate switching (Zhang et al., 2006). This entails creating, for each input to the buggy statement, the output that would lead to test success. The final piece involves synthesizing the repaired expression using input-output component synthesis.

Semfix concentrates its efforts on the right-hand side of assignments and boolean conditionals, aiming to synthesize expressions that integrate both arithmetic and first-order logics. The program synthesis part harnesses the capabilities of the Z3 SMT solver (de Moura and Bjørner, 2008) to efficiently resolve the produced repair constraints.

The efficacy was assessed through a validation involving 90 bugs, in which 48 of them were fixed. In comparison, Genprog, another well-known program repair tool, managed to repair 16 bugs in the same scenario.

Nopol (Xuan et al., 2017) also employs a constraint-based approach and specifically targets conditional bugs. Its focus lies in either modifying existing if-conditions or introducing preconditions before existing code.

The repair process in Nopol also requires a test suite, which is expected to encompass both passing test cases, outlining the expected program behavior, and at least one failing test case that exposes the bug in need of repair.

The fault localization phase in Nopol is orchestrated through angelic debugging, determining the anticipated values of a condition during test execution. Subsequently, runtime trace collection

captures variables and their values, encompassing both primitive data types and object-oriented features like nullness checks. This data forms the foundation for patch generation. The encoded information is then translated as an SMT problem and presented to the Z3 SMT solver. The solution derived from the solver is then transformed into a code patch, featuring either a modified or newly inserted condition.

Nopol's evaluation involved 22 real-world bugs — 16 with faulty if-conditions and 6 with missing preconditions - across two expansive open-source projects: Apache Commons Math and Apache Commons Lang. Nopol successfully repaired 17 bugs: 13 out of 16 bugs with faulty if-conditions and 4 out of 6 bugs with missing preconditions.

The technique was later expanded to address the repair of infinite loops and implemented as a tool called DynaMoth (Durieux and Monperrus, 2016), showcasing its adaptability to diverse fault scenarios.

2.2.4 Mutation-based Repair

Debroy and Wong (2010) introduce an APR approach that makes use of standard mutation operators documented in the mutation testing literature. This technique considers specific mutation operators: the replacement of arithmetic, relational, logical, increment/decrement, or assignment operators by another operator within the same class, along with decision negation in if and while statements.

The fundamental question posed by the authors is whether mutating a faulty program using realistic mutation operators can yield a plausible fix for certain faults. Recognizing the potential computational challenges posed by mutant generation and execution, they contemplate the significant cost and the explosive number of possible mutants for a given program. To address this, and in line with what is conventionally done in other works, the authors leverage the rankings provided by a fault localizer to guide the mutation process. This strategy aims to mutate program components, such as statements, in the order of their likelihood to contain faults, streamlining the search for a potential fix.

The fault isolation step employs the Tarantula spectrum-based fault localization technique to identify potentially faulty statements. Results indicate a reduction in overhead, ranging from 16.67% to 50% of the total possible number of mutants, by utilizing a fault localizer to prioritize

and filter the lines of code subjected to mutation.

Validation of the technique unfolds across 7 benchmark programs within the Siemens suite and the Ant project. This evaluation, encompassing a total of 135 faults, culminates in a 18.52% repair rate, with 25 successfully fixed faults. The authors offer nuanced insights into the experimental design, emphasizing the trade-off between effectiveness and efficiency in selecting mutation operators. While a broader set maximizes fault coverage, a more focused set ensures efficiency by minimizing overhead, albeit at the expense of addressing certain fault types.

Astor, a publicly available program repair library for Java programs, contains an adapted implementation of Debroy and Wong (2010)'s work for this target language. Furthermore, Astor implements two other repair approaches: jGenProg2, which is based on Genprog, and jKali which only performs code removal.

The Defects4J benchmark was used for Astor's evaluation, which encompasses 224 bugs. Astor successfully repairs 33 out of the 224 bugs, showcasing a repair rate of 14.7%. Alone, jMutRepair is able to fix 17 faults. Furthermore, jMutRepair stands out as the sole strategy to achieve success in rectifying at least one fault within the Apache Lang project.

Rothenberg and Grumberg (2016) introduce a distinctive program repair methodology with AllRepair, shifting the focus to the earlier stages of development. Unlike prevalent automated repair tools designed for scalability and targeted at deployed software, AllRepair aims to be a user-friendly solution applicable in the initial debugging phases. The authors envision its utility for independent programmers dealing with small code snippets and as a routine tool that developers can effortlessly integrate into their workflow immediately after making program changes.

Notably, AllRepair embraces a non-scalable approach, prioritizing simplicity and rapid execution. It operates under the premise that even a non-scalable automated repair method can significantly save time and alleviate frustration during the early stages of development. The tool makes no assumptions about the number of mutations required for repair, accommodating multiple co-dependent buggy locations in its repair solutions.

AllRepair's approach involves turning the program into a set of SMT constraints. The authors make the observation that changing something in the program (mutation) is equivalent to changing one of these constraints. As such, instead of exploring all possible mutations, AllRepair looks for the smallest unsatisfiable set of constraints. This way, it efficiently explores and finds ways to

fix the program.

Evaluation is performed on the TCAS benchmarks from the Siemens suite, featuring 41 faulty versions. Two levels of granularity are explored: level 2 utilizes all mutation operators and level 1 employs a subset. While level 2 fixes 11 bugs efficiently, level 1 introduces a trade-off between repairability and runtime, with a noticeable increase in the number of fixed faults to 18 at the cost of slightly extended execution times: 48 seconds on average compared to 2.3 seconds.

2.2.5 Deep Learning-based Repair

While the APR approaches presented so far have made significant strides in automatically addressing software bugs, they are not without their inherent limitations. As discussed in this thesis, challenges include the production of patches acceptable to programmers, the risk of overfitting to specific test cases in generate-and-validate techniques, and the observation that some methods might seemingly achieve repair by predominantly removing pieces of functionality (Qi et al., 2015). Additionally, existing approaches often exhibit limitations in scope, focusing solely on specific types of bugs, relying on handcrafted rules or templates, and struggling with the handling of long-range dependencies within code.

To tackle these challenges, researchers have explored leveraging project history. Approaches like that of Le et al. (2016) delve into the past history of projects, examining bug-fix patches and comparing them to automatically generated patches. Similarities to patches found in the historical data of mined projects are deemed more relevant, increasing the effectiveness of repair strategies. Another notable approach, Prophet (Long and Rinard, 2016a), identifies patches from past fixes by first localizing likely faulty code through test case execution and then generating patches from correct code using a probabilistic model.

The significance of utilizing software development history, abundantly available in code repositories like GitHub, cannot be overstated. These repositories provide extensive change history and bug-fixing commits from diverse software projects. Deep learning techniques can leverage this rich dataset to learn from large software engineering repositories.

Neural Machine Translation

In this section, we will be looking into Neural Machine Translation (NMT) techniques. This approach not only allows for insights into bug-fixing activities in real-world scenarios but also empowers the emulation of genuine patches crafted by developers. Under this scenario, the repair task is interpreted as a translation task in which the languages involved are ”buggy” and ”fixed”, thus harnessing the power of NMT to “translate” buggy code into fixed code. As we explore the area of NMT for program repair, the potential to overcome existing limitations becomes increasingly promising.

Learning Patches via NMT Tufano et al. (2019b) pioneered the use of NMT to learn bug-fixes and automatically generate patches for buggy code, thus introducing a novel approach to APR. Their method is based on three critical observations. Firstly, it highlights the effective utilization of past project histories to discern meaningful program repair patches. Secondly, it addresses the limitations of traditional APR approaches, which often rely on a manually-crafted and somewhat restricted set of transformations or fixing patterns, necessitating significant manual effort and expertise. Lastly, the study aligns with the broader trend in research showcasing the potential of advanced machine learning techniques, specifically deep learning, to extract insights from extensive software engineering datasets.

For bug-fix mining, the researchers analyzed GitHub events from March 2011 to October 2017, identifying commits using specific message patterns suggesting fault-addressing actions. This process yielded 10,056,052 bug-fixing commits, with a 97.6% true positive rate verified through a manual sample analysis. The researchers then extracted source code before and after the fix, discarding non-Java files and new files as these lacked a buggy version for learning purposes. Additionally, commits impacting more than five Java files were also discarded to focus on acquiring specific and localized bug-fixes. This selection resulted in 787,178 bug-fix pairs (BFPs), which formed the dataset for further analysis.

The process to extract information from BFPs involves utilizing the GumTree Spoon AST Diff tool to compute the AST differences between the buggy and fixed files. This tool identifies the sequence of edit actions performed at the AST level, which leads to the transformation of the buggy file’s AST into the fixed file’s AST. GumTree Diff recognizes four edit action types: update,

insert, delete, and move. The analysis considers the set of AST edit actions as defined by GumTree Diff, and the outcome is a triplet comprising the buggy method, the fixed version, and the tree edit actions necessary to convert the buggy code into the fixed code.

The next step involves abstracting the source code of the BFPs into a form more suitable for learning. This process begins by using a Java Lexer and Parser to tokenize the source code, creating a sequence of tokens. Subsequently, an AST representation of the source code is employed to capture its structural aspects. Throughout the abstraction, the researchers retain frequent identifiers and literals, referred to as idioms within the representation. The outcome of this phase includes the abstracted BFPs and the corresponding mapping to original values, facilitating the reconstruction of the original source code. The goal of the abstraction process is to simplify the source code's complexity while maintaining crucial information necessary for learning bug-fixing patterns.

The evaluation involved the creation of two datasets categorizing fixes for small and medium methods. Subsequently, an encoder-decoder model was employed for each set to learn the transformation of buggy code into corresponding fixed versions. The trained models demonstrated the ability to generate patches for unseen buggy code. The NMT-based model trained on small BFPs achieved developer-inspired fixes for 9.22%–50.16% of bugs, while the model trained on medium BFPs produced fixes for 3.22%–28.55% of bugs, with the number of successful fixes increasing when more candidate patches are generated by the models, also called the beam size. The generated patches exhibited high syntactic correctness, ranging between 99% and 82%. Although the models learned to apply a subset of AST operation types used by developers to fix bugs in the test set, these learned operations represent the most crucial ones, theoretically enabling the repair of a significant percentage of bugs.

After around 15 hours of training, the model is able to consistently and efficiently generate 50 candidate patches for a single bug in under a second. This one-time cost was considered acceptable for building a cross-project bug-fixing model. Despite a slight increase in average time per bug with larger beam sizes, ranging from 0.006s (1 patch) to 0.226s (30 patches), the average time per patch generated stayed well below 0.030s.

It's important to highlight that AST edit actions are excluded from the training process; the models do not have access to this information. These actions serve two main purposes: first, to filter bug-fix pairs during consideration, ensuring pairs with excessive or no tree edits are excluded.

Second, they are employed to evaluate the model's ability to emulate frequent edits required for bug fixes. The authors measure the model's performance by comparing the set of edit actions in fixed bugs against those in the entire test set, defining this overlap as "theoretical bug coverage." The model demonstrates robust performance, achieving theoretical bug coverage of 94% and 84% for the small and medium sets, respectively (assuming a beam size of 50).

Tufano et al.'s pioneering work in NMT-based program repair stands out as one of the first significantly impactful initiatives that paved the way for a fusion between historical project knowledge, streamlined repair methodologies, and the capabilities of machine learning, thereby contributing to the transformative evolution of program repair techniques.

Sequence-to-sequence Translation to Locate and Fix Faults Typical APR approaches often assume pre-identified faulty locations or incorporate external fault localization techniques. Moreover, these techniques primarily target logical errors, assuming the code compiles successfully. DeepFix, an end-to-end solution, abdicates of external methods to locate errors. Instead, its neural network is trained to predict both the location and the correction for erroneous statements, making it an iterative solution capable of addressing multiple errors in a single program. Another point that sets DeepFix apart is its focus on what it calls "common programming errors," which it defines as those responsible for compilation failures or build errors. Unlike similar approaches tied to specific programming tasks, DeepFix addresses syntax and structural issues applicable across various programming contexts, analogous to grammatical errors in natural languages. At first, one might think that these common mistakes are almost exclusive to novice programmers. However, it has been observed that experienced developers can inadvertently introduce them into their code too.

DeepFix is based on a variant (Vinyals et al. 2015) of the sequence-to-sequence model (Sutskever, Vinyals, and Le 2014) taking advantage of an attention mechanism (Bahdanau, Cho, and Bengio 2014). Being a "sequence-to-sequence" model, it means it's designed to handle a sequence of data, understand it, and transform it into another sequence of data. More precisely, we are dealing with sequences of code in this case. As such, we give it the buggy code (input sequence) and expect it to produce the corrected code (output sequence). The "attention mechanism" is a feature in the model that helps it focus on specific parts of the input sequence when generating the output sequence, making it more effective at understanding and fixing code.

The program representation in DeepFix encompasses a range of tokens, including types, keywords, special characters, functions, literals, and variables. For types, keywords, special characters, and library functions, a fixed-size pool of names is defined in order to create a shared vocabulary. Each program receives a unique encoding map by randomly assigning distinct identifiers to names in the pool, maintaining semantics and reversibility. Literal values are mapped to special tokens based on types, and the exact values of literals are non-essential. In line with similar techniques, there is an <eos> token denoting the end of a sequence. A program is represented as a sequence of tokens (X), and to enhance prediction accuracy, line numbers are incorporated. Each statement (S) at line (L) is represented as (l, s), forming a program representation of (l1, s1), ..., (lk, sk) <eos>, with l1 to lk as line numbers and s1 to sk as token sequences. The neural network is trained to predict a fix, a smaller output comprising a line number (li) and an associated statement (s'i) that corrects errors in the original statement (si). Making the output sequence smaller than the input sequence facilitates the prediction task.

Overall, DeepFix introduces a robust strategy for addressing multiple errors in a program through an iterative process. The network predicts a single line fix for a tokenized input program, and an oracle evaluates and updates the program accordingly, either accepting or rejecting the fix based on whether the compiler does not produce more errors and other heuristics. The iterative approach continues until the program is error-free, the network deems it correct, the fix is rejected, or a predetermined iteration limit is reached. The network's actions include potential line insertions, deletions, and substitutions, and the oracle ensures successful program text reconstruction via the encoding map previously mentioned.

DeepFix's repair strategy offers several advantages. Firstly, it takes a comprehensive approach by showing the entire program to the network, helping it analyze and fix programming errors globally. The network's ability to focus on different parts of the program allows it to understand the structure and syntax, making predictions more accurate. Including line numbers in both input and output simplifies the prediction task. DeepFix can also fix multiple errors in a program through an iterative process. The oracle, an important part of the strategy, keeps track of progress and prevents unnecessary changes during fixing. Additionally, the strategy is versatile and can handle various error types. For example, it can address logical errors by using a test engine and test suite as an oracle, accepting fixes that make the program pass more tests.

In terms of fault localization, DeepFix is able to autonomously localizes errors, achieving a

78.68% accuracy in identifying 7262 out of 9230 erroneous lines in the first iteration. Utilizing beam search, the network's top-5 predictions include 87.50% of the erroneous lines. This means that beyond the primary goal of predicting fixes, DeepFix can also assist programmers by reporting suspicious lines.

DeepFix achieves a comprehensive success rate, fully fixing 27% and partially fixing 19% of the 6971 erroneous programs. It effectively resolves 32% (5366) of the originally generated 16743 error messages. Additionally, DeepFix demonstrates efficiency, taking only a few tens of milliseconds on average to rectify errors in a test program. When applied to a seeded dataset with known expected fixes, DeepFix fully fixes 56%, partially fixes 17%, and successfully resolves 63% of the compilation error messages. Indeed, DeepFix fixes substantially more programs in the seeded dataset than the raw dataset. Comprehensively, the authors created that particular seeded dataset to enable a more controlled analysis of the results and better understand the limitations and areas they need to work on the approach. Nonetheless, this indicates that the dataset in question does not reflect all possible errors from the raw dataset with the authors acknowledging there is still room to improve in this regard.

Unlimited Vocabulary and Long-range Dependencies SequenceR is also an end-to-end method for fixing program errors based on sequence-to-sequence learning. As already noted, working with code comes with unique challenges. The raw data is inherently noisy, demanding careful curation efforts to sift through commits and pinpoint those addressing specific tasks. Unlike natural language, programming languages exhibit a low tolerance for the misuse of rare words, such as identifiers and numbers. The compilers or interpreters in programming are unforgiving, requiring a great deal of precision when using the language. Plus, in code, the range of dependencies connecting different parts of it can stretch over longer distances compared to normal language; variables declared dozens of lines before or even in completely separate modules might very well present a great impact, a characteristic distinct from the shorter-range dependencies typical in natural language, in which associated words are often used in the same sentence of particularly close to each other.

SequenceR tackles these problems by first focusing on one-line fixes: it predicts the corrected version for a buggy line. To make this work, the researchers build a dataset of one-line commits for training and testing. What sets SequenceR apart is its sequence-to-sequence network archi-

tecture, specifically designed to handle challenges like the unlimited vocabulary and long-range dependencies problems. Indeed, a common problem with sequence generation is that only tokens which are in the training set are available for output — vocabulary problem. To tackle this, the authors use the copy mechanism. The basic intuition behind it is that words not available in the vocabulary may be directly copied from the input sentence over to the output translated sentence. Ultimately, this allows SequenceR to predict fixes even if they contain rare tokens not explicitly in its vocabulary. This is particularly helpful for dealing with uncommon API calls or identifiers. To address the long-range dependency problem, the team constructs an abstract buggy context from the buggy class, capturing crucial context around the buggy code and simplifying the input sequence's complexity. This approach allows SequenceR to handle the lengthy connections needed for effective fixes.

SequenceR specializes in replacing existing source code, omitting scenarios involving line deletions due to compatibility issues with sequence generation and excluding additions because spectrum-based fault localization, commonly used in related works, proves ineffective in such cases. Leveraging fault localization techniques, SequenceR identifies problematic methods and lines, proceeding to organize this data through a buggy context abstraction process. The challenge lies in balancing the need for a concise sequence of tokens, given the limitations of sequence-to-sequence models with long sentences, while preserving as much relevant information as possible.

In each bug location, SequenceR includes the buggy line, marked by special tokens indicating its start and end to propagate this information to the model, the buggy method to provide insights into its interaction with the rest of the method, and the buggy class containing only instance variables, initializers, constructor signature, and non-buggy method signatures. SequenceR utilizes truncation to control the size of the abstract buggy context, ensuring efficient processing of input files with varied lengths. This involves choosing a truncation size, minimizing its use, including as many tokens as possible from the buggy line within the limit, and maintaining a ratio of twice as many tokens before the buggy line as after it.

The abstract buggy context, used as input for the sequence-to-sequence network predicting the fixed line, is internally represented with a token sequence from the vocabulary, substituting out-of-vocabulary tokens, i.e. unknown tokens, with the "<unk>" token. During patch inference, the abstract buggy context is still generated, but beam search is employed to produce multiple likely patches for the same buggy line. SequenceR's accuracy, tested on CodRep4 and Co-

dRep4Medium, is 344/1,116 (30.8%), outperforming Tufano et al.'s 157/1,116 (14.1%). This highlights the efficacy of constructing the abstract buggy context with the copy mechanism for higher accuracy. For an external evaluation, the authors tested SequenceR on 75 Defects4J bugs fixed by human developers using one-line patches. SequenceR successfully generated patches for 58 bugs, with 53 having at least one compilable patch, 19 passing all tests (plausible), and 14 deemed correctly fixed, being semantically identical to the human-written patches. Notably, in 12 out of these 14 cases, the patch with the highest ranking in the beam search results coincided with the semantically correct solution.

Some of the limitations with SequenceR include only addressing bugs inside a method, focusing on one-line fixes, and lacking an iterative process like Tufano's work. Moreover, while long-range dependencies are acknowledged, they are limited to class-level, potentially overlooking complex connections in software projects.

Context Learning and Transformation Learning NMT-based approaches encounter three significant challenges when applied to APR. Firstly, they treat APR as a translation task, lacking explicit knowledge of modified parts in the faulty code during training. This requires the models to implicitly learn alignment, leading to potential imprecision in identifying fixing locations in new faulty code. Secondly, the sequence-based representation struggles to capture the well-defined syntax and semantics of source code, risking imprecise mappings and incorrect fixes due to the implicit recovery of code structure. Lastly, handling the context of code around fixing locations proves challenging, with noise introduced by using entire methods and insufficient context limiting the accuracy of derived fixes.

With this in mind, DLFix (Li et al., 2020) was introduced as a two-layer tree-based DL model. DLFix targets APR as a code transformation learning challenge. It diverges from conventional sequence-to-sequence NMT approaches by utilizing a tree-based Recurrent Neural Network (RNN) with the AST representing the source code. More specifically, DLFix introduces a separation between the learning of the context surrounding the code and the learning of bug-fixing code transformations. This separation is implemented through the incorporation of two distinct layers.

For context learning, the AST of a buggy method is subject to a process where the identified buggy sub-tree is replaced by a summarized node, encapsulating its structural details. This, along with the non-buggy AST sub-trees, forms the contextual information fed into the context learning

layer. The output from this layer is a vector that precisely encapsulates the surrounding context. In the subsequent code transformation learning step, a distinct tree-based RNN model is trained using the changed sub-tree before and after the fix. The context transformation vector, derived in the earlier context learning layer, is integrated as an additional input. DLFix's separation between context and transformation learning enhances its understanding of surrounding code by focusing on aligning changed sub-trees, avoiding incorrect alignments and acquiring more fine-grained transformations during training.

DLFix underwent evaluation on two established bug datasets, Defects4J and Bugs.jar, as well as a novel dataset encompassing over 20,000 real-world bugs from eight large Java projects. In contrast to 13 pattern-based APR tools, DLFix surpassed eleven and exhibited compatibility with the top two tools. Furthermore, when compared against four state-of-the-art DL-based APR approaches (Hata et al., 2018; Chakraborty et al., 2022; Tufano et al., 2018, 2019a), DLFix outperformed them, identifying 2.5 times more bugs than the best-performing baseline and 19.8 times more bugs than the worst-performing baseline.

Convolutions and Ensemble Learning State-of-the-art techniques have limited success in locating correct patches within their search spaces (Long and Rinard, 2016b). CoCoNuT (Lutellier et al., 2020) is a context-aware NMT architecture that has two separate encoders: one for the buggy lines, the other one for the context, similar to DLFix but without representing source code as an AST. However, it uses ensemble learning on a combination of convolutional neural networks (CNNs) instead of RNNs. This may seem like an intricate technical aspect; but is what sets this technique apart. Ensemble learning is used to address the diversity of bugs and fixes, as it allows combining models with different levels of complexity, capturing various relations between buggy and clean code. This enables the approach to learn diverse repair strategies suited for different types of bugs. As for the use of CNNs, the authors justify it due to a specific difference between natural language and source code. Unlike natural language, which is read sequentially from left to right, source code is not executed sequentially in the same manner. Relevant information may be situated farther away from the buggy location, posing a challenge for traditional RNNs and LSTM layers. To tackle this, the authors propose an architecture that leverages CNN layers. These layers capture relations at different levels, addressing both short-term dependencies within statements and long-term dependencies within functions.

As we say when discussing other NMT-based approaches, effectively representing context in bug-fixing tasks poses a challenge for NMT models, with concatenation of context and buggy code leading to long input sequences, limiting applicability to short methods (Chen et al., 2021; Tufano et al., 2018). Furthermore, it also hinders the extraction of meaningful relations between tokens primarily due to the addition of noise from excessive context. In contrast to natural language, source code's larger vocabulary, including infrequent tokens and the significance of letter case, requires practitioners to reduce the vocabulary size. The authors adopt a tailored tokenization approach akin to word-level tokenization but tailored to programming languages, separating operators and variables without space. Enhancements include considering underscores, camelcase, and numbers as separators. A new token (<CAMEL>) marks camel case splits. String and number literals are abstracted, excluding common numbers (0 and 1) from the training set. This minimizes out-of-vocabulary tokens to less than 2% in benchmarks.

In an evaluation against 27 APR techniques across six bug benchmarks in four programming languages (Java, C, Python, and JavaScript), CoCoNuT successfully fixed 509 bugs, notably addressing 309 bugs not remedied by other tools. Its effectiveness stems from learning new patterns, extracting donor code from the bug's context, and leveraging historical training data. The incorporation of ensemble learning significantly enhances CoCoNuT's performance, demonstrating a 50% improvement when employing 10 models compared to a single model. In terms of efficiency, with 10 models, CoCoNuT averages 16.7 minutes to generate 20,000 patches for one bug, while a single model achieves this in an average of 1.8 minutes.

Pre-trained Language Models

Even though translation-based techniques have shown good results and proved to be promising research paths to explore for APR, some inherent limitations still affect their performance. Notably, their search space frequently lacks the correct fix. Moreover, their strategy overlooks strict code syntax. Ultimately, these drawbacks hinder both their effectiveness, by preventing such approaches to even reach a useful patch, and also their efficiency, by generating numerous uncompilable patches overly often. Despite surpassing manual rule/template crafting techniques in certain aspects, translation-based methods still exhibit equitability or introduce new drawbacks in specific scenarios.

In natural language, pre-trained language models have significantly enhanced various NLP

tasks, leveraging the learned probability distribution over word sequences from extensive text data. These models offer the flexibility of fine-tuning for specific tasks or direct out-of-the-box use. Because the training process is based on unsupervised learning, which means it does need any ground truth or data labeling, a pre-trained language model is typically trained on a very large dataset. This way, it is able to gather more information regarding sentence structure (syntax) and about what human-like text is (readability).

The effectiveness of these models is often amplified by their sheer size, allowing them to capture intricate patterns and nuances in language. For this reason, it is common to refer to them as large language models (LLMs). Overall, this improves the quality of the generated text. Specifically, the application of LLMs for programming languages has allowed for a significant enhancement in code understanding and generation. These capabilities allow for a great deal of automation in code-related tasks. In this section, and in line with this thesis, we will focus on the use of LLMs for program repair.

Separating PL Learning from Patch Learning CURE (Jiang et al., 2021) is a technique that does not separate itself entirely from NMT-based approaches. Yet, it recognizes the inadequacies of targeting the APR problem by implementing a translation-based approach and suggests integrating the use of a pre-trained language model. Afterwards, this model is specialized for the APR task by fine-tuning it, thus building what the authors call an APR model.

First, to address the challenges of learning developer-like source code, the authors train a language model on open-source programs. This improves the probability of a sequence of tokens being a real-world code snippet. Specifically, CURE uses a Generative Pre-trained Transformer (GPT) (Radford et al., 2018) for their language model. GPTs are a type of LLM that have shown tremendous improvements in many NLP tasks (Radford et al., 2019). The pre-training of this LLM is an essential part of CURE's approach. It allows for the separation of programming language learning and patch learning. The authors mention two advantages with this. First, it uses unsupervised learning; therefore one can extract a large amount of data automatically and train it. Second, during fine-tuning, the APR model already knows the syntax, making the fine-tuning more efficient.

After pre-training the first model, CURE fine-tunes it for the APR task. This is done by combining the pre-trained model with an NMT model as the APR model. The APR model takes buggy

lines and their context as input and aims to generate a correct patch. During the fine-tuning process, the APR model is trained to learn the transformation from the buggy lines and context to the correct fix.

As for the tokenizer, CURE uses byte-pair encoding (BPE) to tokenize compound words and rare words. BPE still allows for the flexibility of considering camel letters, underscores, and numbers to split long identifiers while further addressing the OOV problem.

Indeed, CURE is very similar in nature to the NMT-based approaches presented previously. However, by explicitly separating the learning process of learning how to write source code, without the end goal of repairing programs in mind, and the learning process of producing correct patches, which is common to other translation techniques, CURE aims to mitigate the negative effects that having that whole part in a single step presents.

For evaluation, two widely-used benchmarks, Defects4J and QuixBugs were used. Patched projects are compiled and test suites are executed to find plausible patches, i.e., patches that pass the relevant test cases. Two co-authors independently checked plausible patches and considered as correct patches only those that are equivalent to developers' patches. CURE fixes the most number of bugs, generating plausible patches for 104 Defects4J bugs, in which 57 were correctly fixed, and generating plausible patches for 35 QuixBugs bugs, in which 26 were correctly fixed. According to the paper, CURE outperforms all the APR tools in was compared with.

There is one bug for which CURE is the only approach able to fix it:

```
1 - Object clone = createCopy(0, getItemCount() - 1);
2 + Object clone = createCopy(0, Math.max(0, getItemCount() - 1));
```

The correct fix requires ensuring the second parameter is non-negative. Pattern-based approaches fail to produce a correct patch because they implement no patterns or combination of patterns that recreates the necessary transformation. NMT-based approaches fail to fix it as they are dependent on having patches that recreate similar scenarios. In fact, the authors state that in their patch training data consisting of 2.72 million training instances there are only two similar fixes. This makes it virtually impossible for NMT-based approaches to capture this transformation due to the lack of representation. However, ensuring non-negativeness is a common Java coding practice and the pre-trained language model, which is not coupled to any particular repair task, captures that.

Considering the top-30 patches produced for each bug, CURE generates compilable patches (39%) more often that SequenceR (33%) and CoCoNuT (24%).

Repairing with Out-of-the-box Infilling We have noted that the context in which a bug occurs is crucial to fix it. Such context is usually represented by source code that is immediately near the bug or by elements, such as variables, that may be defined further away but are still referred to in the buggy area. This context provides extremely useful syntactic and semantic information.

Many of the learning-based techniques explored so far in this thesis have this in mind by clearly pointing out their efforts in extracting relevant context in order to associate it to the buggy code they intend to fix. Generally speaking, those techniques combine such context and the associated buggy code as plain text (Jiang et al., 2021; Lutellier et al., 2020) or through more structured representations (Li et al., 2020; Zhu et al., 2021). AlphaRepair (Xia and Zhang, 2022) argues that this process in unnatural. The basis for the argument is that it is challenging for the models to distinguish the patch location within the context, or effectively merge the separate bug and context encodings. Indeed, in light of what we have been discussing so far, this makes sense. We previously saw that we are limiting APR's potential by forcing models to learn how to write source code while teaching them how to write patches. By seeing these as two separate steps: language learning and patch learning, we allow APR to break free from the boundary imposed by inadvertently merging those two.

Still, we have to deal with the strong focus that many learning-based APR approaches place on training, either by creating NMT models from scratch or by fine-tuning existing pre-trained models. AlphaRepair recognizes this issue and deviates from modeling what a repair edit should look like and, instead, directly predicts what the correct code is based on the context information. They do this by leveraging the out-of-the-box infilling capabilities of CodeBERT (Feng et al., 2020), a pre-trained language model for multiple programming languages. Infilling is the capability of predicting or completing missing code elements within a given code snippet. The key aspect is that the surrounding context can be used and we are not limited to what is before, that is, on the left. CodeBERT uses the masked language modeling (MLM) objective, which is what enables it to perform infilling. During training, MLM randomly masks out tokens and the model is trained to recover them. During inference, this allows the model to operate via zero-shot learning,

which essentially means filling in blanks (in the form of masked tokens) without receiving explicit examples in the prompt.

First, AlphaRepair replaces each potentially buggy line with a mask line. Note that, unlike in the MLM training objective, tokens are not randomly masked out. A mask line is a line with one or more mask tokens — <mask>. AlphaRepair uses 3 strategies to produce mask lines. The simplest strategy is to replace the entire buggy line with a line containing only mask tokens — line replacement — and to insert mask lines where only mask tokens are added before or after the buggy line — line insertion. Another strategy for generating mask lines is by reusing partial code from the buggy line. The buggy line is first separated into its individual tokens and then the last/first token is kept while replacing all other tokens before/after with mask tokens. This process is repeated by appending more tokens from the original buggy line to generate all the possible versions of partial mask lines. Finally, several template-based mask line generation strategies targeting conditional and method invocation statements were implemented as they are two of the most common bug patterns (Le et al., 2016; Xuan et al., 2017; Durieux and Monperrus, 2016; DeMarco et al., 2014).

CodeBERT is then queried to fill the mask line with replacement tokens to produce candidate patches for a buggy program input. This is where zero-shot learning is performed (with no fine-tuning). The bidirectional nature of CodeBERT also allows to capture both the contexts before and after the buggy location for effective patch generation. A key difference between AlphaRepair's input and the MLM objective used for training is that in AlphaRepair the masked tokens are grouped together, which means there may be multiple mask tokens contiguously, the immediate context before and after are masked tokens. In contrast, during training, the masked tokens are spread out and each of them has sufficient context tokens before and after. To circumvent this, AlphaRepair uses an iterative process to produce candidate patches. In the initial input, certain tokens are masked and CodeBERT is used to predict the top N most likely replacements for the first masked token. N represents the beam width which allows for multiple possibilities. In the next iteration, AlphaRepair replaces the first masked token with the top N replacements from the previous step and seeks token pairs (first and current masked token) with the highest joint conditional probability. This joint score is then used to retain the top N generated token sequences and the process continues until tokens are generated for all the masked positions in the input. This joint score is temporary as it is conditioned on the mask tokens whose values have

not yet been decided and the conditional probability does not account for their future concrete values. Under this approach, CodeBERT has at least one side of the immediate context — the left side. This is similar to what is commonly used in code generation tasks which can only access the context on the left. But for this case, it is only an approximation as the true probability depends on the tokens to the right, which are still masked and yet to be predicted. To address this issue, AlphaRepair re-ranks the fully generated N candidate patches. Each originally masked token (now with a concrete value) in each patch in masked again and CodeBERT is queried to obtain the conditional probability of that token. The same process is applied for all other previous mask token locations and the true joint score is computed given both contexts before and after.

Patches are validated by compiling them and, if successful, executing the test suite. AlphaRepair is evaluated using the Defects4J and QuixBugs benchmarks. Furthermore, for Defects4J, the evaluation considers two scenarios: perfect and non-perfect fault localization. For Defects4J under the perfect fault localization scenario, AlphaRepair can successfully generate correct fixes for 74 bugs and outperforms all the learning-based and traditional APR approaches it was compared with. Also for Defects4J but instead considering a non-perfect fault localization, AlphaRepair is able to produce correct patches for 50 bugs. For this last scenario, because there is no ground truth for the suspicious line, the beam width is lowered from 25 to 5 and only the top-40 ranked lines are considered. As such, less bugs — 50 instead of 74 — are fixed but state-of-the-art tools are still outperformed. For QuixBugs, AlphaRepair fixes 28 and 27 bugs for Java and Python, respectively, and outperforms the techniques it was compared with.

Type-awareness in DL-based repair Generating untypable patches is transversal to program repair techniques. Although DL-based APR approaches are able to learn a programming language's syntax very effectively, the learning process does not explicitly take into account type information. In fact, the repair process as a whole tends to ignore this aspect, or at least takes it very lightly. Normally, this concern is moved (or postponed) to a validation step, in which patches are discarded if they fail to compile. As a result, APR approaches often generate patches that are syntactically correct but violate typing rules, creating patches that are not executable or that introduce new bugs.

In this sense, Tare (Zhu et al., 2023) aims to improve the effectiveness of APR approaches by taking into account typing rules. The way Tare addresses this limitation is by incorporating type-

awareness into the repair process, resulting in more accurate and typable patches. Specifically, Tare trains the model to acquire a set of typing rules, effectively learning the inherent constraints.

This approach introduces three components:

- T-Grammar is a type of grammar that integrates type information into a standard grammar. Each non-terminal symbol in the grammar is assigned a type, resulting in a set of new symbols that refine the original grammar. For example, instead of the production rule $Exp \rightarrow Exp + Exp$, T-Grammar has the production rules such as $Exp\_Numeric \rightarrow Exp\_Numeric + Exp\_Numeric$ and $Exp\_String \rightarrow Exp\_String + Exp\_String$. This allows the neural network to predict not only the grammar rule, but also its types, enabling the construction of T-Graph from partial programs.
- T-Graph is a representation of code that integrates the key information needed for type checking an AST. In T-Graph, each part of the code is represented by a node, and the relationships between the parts of the code are represented by edges between the nodes. The edges are labeled with the types of the relationships between the nodes.
- Tare is the type-aware neural program repair approach that encodes the T-Graph and generates the patches guided by T-Grammar. It is built upon Recoder, one of the state-of-the-art DL-based APR approaches that also uses a structural representation for code, instead of plain text like commonly done in other works. Tare changes the grammar in Recoder into a T-Grammar and replaces the neural components of Recoder encoding ASTs with neural components encoding T-Graphs.

In a comparative study, Tare outperformed 5 DL-based APR approaches: CoCoNuT (Lutellier et al., 2020), CURE (Jiang et al., 2021), RewardRepair (Ye et al., 2022), DLFix (Li et al., 2020), and Recoder (Zhu et al., 2021). For the Defects4J v1.2 benchmark, Tare repaired 62 out of 393 bugs (15.8%) and generates compilable patches 54.6% of times considering the top-30 patches. It is questionable if Tare generalizes well for other programs, as for the new 444 bugs introduced in Defects4J v2.0, it repairs 32 of them (7.2%). For Quixbugs, Tare repairs 27 out of 40 bugs (67.5%). However, Quixbugs is a dataset with much smaller and simpler bugs.

As we mentioned, Tare is built on Recoder, a technique by many of the same authors. The key contribution in Recoder is a syntax-guided process which integrates the grammatical constraints but not typing relations. Indeed, Tare is not the first APR approach to try to incorporate type information in the repair process. RewardRepair introduces a semantic training approach to

help neural models learn the corresponding knowledge via backpropagation. That is, when the model generates an uncompilable patch, RewardRepair punishes the candidate via decreasing the reward during training. On the contrary, Tare directly encodes the knowledge into the encoder. The authors argue that encoding the typing rules in the encoder directly is more effective because the model estimates higher probabilities for the compilable patches. Thus, more typable candidates will be preserved in the beam.

While the overall compilable rate experiences a decrease with increasing beam size (k) in Tare, with rates of 54.6% for k=30, 48.6% for k=100, and 46.7% for k=200, the comparative improvement of Tare over other tools shows an upward trend. Specifically, for k=30, Tare exhibits a 9.3% improvement, for k=100, an 11.1% improvement, and for k=200, a 12.5% improvement. This indicates that Tare's effectiveness relative to other tools becomes more pronounced as the beam size increases, which further corroborates the authors argument for encoding typing rules.

On Understanding Contextual Changes of Failures

3

"On Understanding Contextual Changes of Failures"
Francisco Ribeiro, Rui Abreu, and João Saraiva
In: Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security, QRS 2021
Hainan, China, December 06—10, 2021
DOI: **10.1109/QRS54544.2021.00112**

Abstract *Recent studies show that many real-world software faults are due to slight modifications (mutations) to the program. Thus, analyzing transformations made by a developer and associating them with well-known mutation operators can help pinpoint and repair the root cause of failures. This chapter proposes a mutation operator inference technique: given the original program and one of its subsequent forms, it infers which mutation operators would transform the original and produce such a version. Moreover, we implemented this technique as a tool called Morpheus, which analyzes faulty Java programs. We have also validated both the technique and tool by analyzing a repository with* 1753 *modifications for* 20 *different programs, successfully inferring mutation operators* 78% *of times. Furthermore, we also show that several program versions result from not just a single mutation operator but multiple ones. In the end, we resort to real-world case studies to demonstrate the advantages of this approach regarding program repair.*

3.1 Introduction

Software development methodologies have quickly evolved in recent years. Teams of engineers develop complex and large software applications in a collaborative environment, using version control systems, testing frameworks and continuous integration mechanisms to improve productivity. As is increasingly common in such a development environment, a new software feature or

maintenance update needs to be *pushed* (to the version control system) to be available to the development team. After such a push, the continuous integration mechanism performs a set of tests that the new software ought to pass. When all the tests pass, a new software build is produced. However, often the software does not pass such tests because of a bug that was unintentionally introduced. Developers typically look at the source code to identify changes between versions that may have introduced the bug. This analysis is frequently based on *diff* reports and misses the context of the fault. Let us consider an example taken from the *Bugswarm* (Tomassi et al., 2019) repository containing real-world bugs and their fixes:

```
1 132c133
2 < ... property.toUpperCase());
3 ---
4 > ... property.toUpperCase(Locale.ENGLISH));
```

Simply knowing that line 132 got modified does not reflect the modifications' *semantics* (context), making it difficult to understand and fix the problem without concrete clues.

If we analyze further, we see that this modification essentially adds a parameter in an already existing method call. More precisely, `property.toUpperCase()` now gets an additional parameter, represented by the call `property.toUpperCase(Locale.ENGLISH)`. An *argument number change* (Ma et al., 2002; Offutt et al., 2006; Derezin□ska, 2007) is an example of a mutation that changes the number of input parameters in method invocations, given that there is a definition for the same method name which accepts the new arguments.

Although this is a simple motivating example, recent studies show that many real-world software faults are coupled to mutation operators (Just et al., 2014; Andrews et al., 2005; Daran and Thévenod-Fosse, 1996; Namin and Kakarla, 2011).

This chapter presents a mutation operator inference technique that, given the original program and one of its subsequent versions, infers the mutation operators capable of producing such an alternative. This is achieved by interpreting the changes made to the abstract syntax tree (AST). Although the different program versions are obtained by manually modifying the source code, as it happens in a real-world software project, throughout this chapter, we sometimes refer to them as *mutants*, even though there was no mutation testing tool involved.

The contributions of this chapter are:

1. a technique that allows for the detection/inference of mutation operators based on AST

transformations;

2. a tool implementing this technique;
3. a dataset produced by validating our technique and tool over an existing repository of 1753 manually modified programs with information about the detected mutation operators;
4. a repair strategy that reverts the detected mutation operators and a tool implementing it;
5. a case study investigation with real-world bugs from the *Bugswarm* and *Defects4J* repositories showing how this work can benefit program repair.

Being able to infer the mutation operators is a first step to incorporate automated program repair in a continuous integration system, where a faulty program is fixed by considering the *contextual modifications* that led to the introduction of the bug. By analyzing a considerable number of programs, we can verify the most common mutations. As far as we know, there is no ranking that lists each mutation operator's frequency in a real-world scenario. That is, the number of times a manually created version of a program translates to the application of commonly known mutation operators.

While mutation testing focuses on injecting faults as small modifications, our work aims to analyze the prevalence of those exact patterns of modifications in source code modified by humans.

3.2 Mutation Analysis

Mutation operators are one of the pillars of *mutation testing* (Coles et al., 2016), a technique that introduces faults in source code to assess the quality of tests. This quality is measured by evaluating the test suite's ability to detect the mutated programs. That is, tests that cover the mutated code should fail. Mutation testing relies on the quality of mutation operators and their aim is to mimic programming errors, such as using the wrong value for a constant, applying an incorrect binary operator or referring to a wrong variable's name. The alternative programs produced by these operators (called *mutants*) are semantically correct, i.e., the program is valid. Research concerning mutation operators has been widely conducted. As a result, many operators representing precise transformations have been defined. Although the study of mutation operators started by targeting general programming aspects (DeMillo et al., 1978; King and Offutt, 1991), the definition of such operators can be more specific, with some works describing mu-

tations specialized for object-oriented settings (Ma et al., 2002, 2006). The mutation operators considered in this work have been taken from previous literature and incorporated into existing mutation testing tools (Coles et al., 2016; Just, 2014; Ma et al., 2006).

Table 3.1: Mutation operators: possible inferences

Mutation Operator	Example
ConstantReplacement	int i = **0** → int i = **1**
RelationalOperatorReplacement	x **<=** 2 → x **<** 2
VarToVarReplacement	next = **var1** → next = **var2**
StatementDeletion	~~int n = 1;~~
ArithmeticOperatorInsertion	int a = b; → int a = b **+ 1**;
NonVoidMethodDeletion	String s = ~~getName();~~
VarToConsReplacement	int i = **j**; → int i = **0**;
ReturnValue	return **2**; → return **3**;
UnaryOperatorInsertion	setX(x); → setX(x**++**)
ConditionalOperatorReplacement	x<=2 **&&** y<4 → x<=2 **||** y<4
VoidMethodDeletion	~~print("str");~~
ConditionalOperatorDeletion	x<=2 **&&** y<4 → x<=2
ArithmeticOperatorReplacement	float x = a ***** b → float x = a **/** b;
MemberVariableAssignmentDeletion	private int x ~~= 3~~;
AccessorModifierChange	**public** void... → **private** void...
UnaryOperatorReplacement	i**++** → i**- -**
RemoveConditional	if(**x < 2**) → if(**true**)
ArithmeticOperatorDeletion	float x = a ***** b → float x = a;
ConsToVarReplacement	int x = **2**; → int x = **a**;
ConditionalOperatorInsertion	x<=2 → x<=2 **&&** y<4
UnaryOperatorDeletion	setX(x**++**) → setX(x);
ConstructorCallReplacementNull	String s = ~~new String()~~ null;
StaticModifierDeletion	public **static** int... → public int...
AccessorMethodChange	point.**getX**(); → point.**getY**();
BitshiftOperatorReplacement	1 **«** 30 → 1 **»** 30
ReferenceReplacementContent	someObj → someObj**.clone()**
StaticModifierInsertion	public int... → public **static** int...
TrueReturn	return **x<=2**; → return **true**;
FalseReturn	return **x<=2**; → return **false**;
ArgumentTypeChange	method(int x){ → method(long x){
ArgumentNumberChange	new Person(); → new Person(**"joe"**);
BitshiftOperatorDeletion	1 **«** 30 → 1
BitwiseOperatorReplacement	int x = a **|** b; → int x = a **&** b;
Negation	int x = num; → int x = **-**num;

Test coverage is not enough to guarantee a test suite's quality, as it is purely a quantitative measure of the amount of source code we exercise. Having a way of automatically generating alternatives to our programs is helpful, as now we can qualitatively measure if our test suite is

robust enough to react to the presence of bugs. Moreover, interpreting the source code and manually creating mutants of the most pertinent parts of the program's logic would be very inefficient. However, there is still the question of whether artificial faults, i.e., mutants, are a suitable replacement for real faults. Several studies (Just et al., 2014; Andrews et al., 2005; Daran and Thévenod-Fosse, 1996; Namin and Kakarla, 2011) have analyzed the connection between real faults, i.e., *bugs* accidentally introduced while developing a real-world application, and mutants. In general, results obtained by using real errors are also obtained by using mutants. In particular, one of these studies (Just et al., 2014) even shows that real faults are coupled with commonly used mutation operators by mutation testing frameworks. This means that real faults can be translated as the application of well-known mutation operators. Thus, we argue that if we better understand how software was modified in terms of the application of mutation operators, we can efficiently design a repair strategy that fixes the program. The changes introduced by these operators represent small modifications to a program's logic and can be interpreted as changes to the program's *structure*. Therefore, to accurately detect these modifications, we focus on the structural representation of programs. The AST represents a program's source code in which emphasis is given to structure and contents. The nodes composing such trees represent constructs used in the corresponding program, e.g., *if's*, *while's*, *expressions*, etc. Therefore, when obtaining the set of modifications by comparing programs based on their AST's, we can better understand how its structure changed. These modifications are additions, deletions, updates or movements of nodes in the AST. Because these nodes have information related to the source code's context, we can see how a program changed *semantically*.

In terms of AST differencing, Figure 3.1 illustrates the introductory example[1]. As we can see, the area pointed at shows where an *insert operation* was performed. By capturing this change and verifying the context in which it occurs, we can realize its actual meaning. In this case, the change is applied to an invocation and the number of its arguments gets modified. As such, we can infer it translates to applying an `ArgumentNumberChange` operator. Creating a technique that embodies this reasoning allows us to derive the *semantics* of source code modifications automatically. As a result, developers can get support in reasoning why certain changes lead to errors.

[1]Colors represent the action types applied to the AST: Green - Insert; Red - Delete; Yellow - Move; Orange - Update

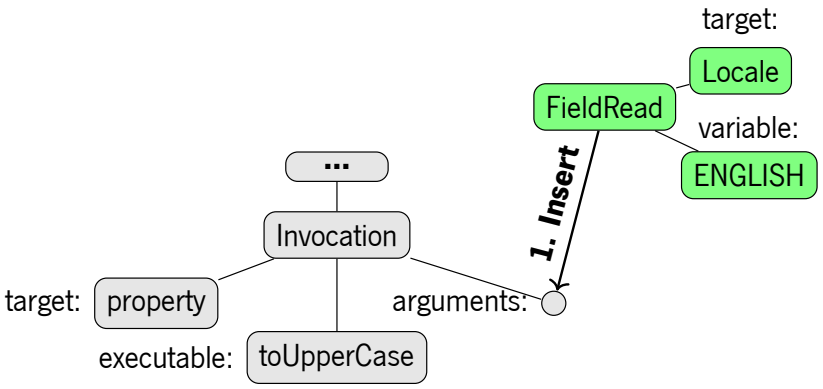


Figure 3.1: Argument number change: introductory example

3.3 Inferring mutations

To infer the mutation operators responsible for a new program version, we focus on the AST representations of both the original and the mutated versions. This approach allows us to circumvent a significant limitation concerning *textual diffs*: we lose the code's structural notion, i.e., what each piece represents and how it connects to the remaining parts of the source code. Furthermore, detecting changes at a textual level regarding the evolution of a file mainly considers two possible representations: insertions and deletions. Detecting changes made to structured data is a topic that has been subject to a considerable amount of research (Chawathe et al., 1996; Fluri et al., 2007; Falleri et al., 2014). This set of changes, called an *edit script*, is computed at a node level and considers more types of transformations: insertions, deletions, updates and moves. In short, given two trees, T_1 and T_2, we are interested in obtaining the edit script, which, when applied to T_1, produces T_2. Consider the following original code and corresponding mutant:

```
1   return h & (length - 1); //original
2   return h & (length);   //mutant
```

In mutation testing, we could obtain the previous mutant by applying the `ArithmeticOperatorDeletion` operator. Detecting such modifications based on the textual representation of this part of the source code would be cumbersome. We only know which lines changed and we have no information about which part of the line was modified. Moreover, we would have to parse a partial program, which is a task subject to ambiguities (Dagenais and Hendren, 2008). There is no sensible way of parsing incomplete code without rapidly falling into errors. At some point, the degree of incompleteness will easily cause the

failure of whatever workaround strategies the parser is using.

In turn, if we provide the two complete programs to an AST *diff* tool[2], we obtain a set of transformations, which Figure 3.2 illustrates.

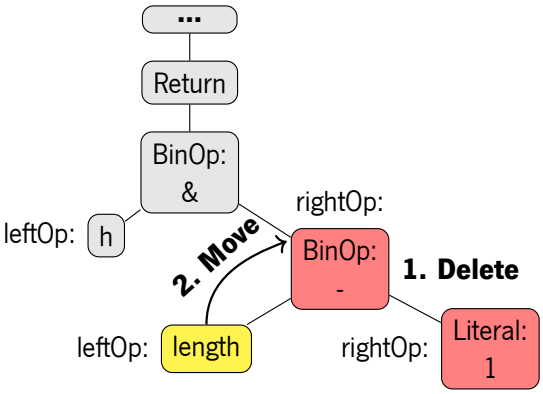


Figure 3.2: Arithmetic operator deletion

Because an AST conveys the program's structure, instead of being directed to a line in the source code, the differencing algorithm analyzes the tree nodes and can detect changes at the component granularity. Also, because tokens are split into their corresponding nodes, carrying their meaning in the source code, we can isolate changes at a more elementary level. Here, a total of two operations are involved:

- a **delete operation** of the node corresponding to the subtraction operator, which in turn implies the deletion of one of its operands, in this case, the literal 1;
- a **move operation** of the leftover operand, which now takes the spot of the deleted one.

However, checking only for the types of operations, in this case, *delete* and *move*, is not sufficient. The simple occurrence of such node modifications can have different meanings, as other mutation operators also manifest themselves through these kinds of AST transformations. In order to correctly pinpoint this case as the application of an *arithmetic operator deletion*, we need:

- to check if the **deleted node** is a binary operator and if it is an arithmetic operator (in this case: "-");
- to check if the **moved node** is one of the operands of the deleted node (in this case: "length").

Of course, when we only have the difference between ASTs, we do not know which mutation operators are present. As such, we need to check against all possible operators until we exhaust

[2] `https://github.com/SpoonLabs/gumtree-spoon-ast-diff`

all options. Similar to what we previously described regarding the *arithmetic operator deletion*, each mutation operator is associated with a set of rules that must be observed to establish its presence accurately.

3.3.1 Multiple mutations

A file may be modified in a way that reflects several mutation operator applications at once. Jia and Harman (2009) define this concept as *higher-order mutants*, referring to them as the injection of two or more simple faults, which they call *first-order mutants*. However, the AST *diff* does not differentiate groups of transformations and it produces a set with all transformations. We subdivide this set into subsets so that we can better isolate the occurrences of operators.

Overlapping mutations

Let us consider the textual diff:

```
1 173c173
2 < if (inflection.match(word)) {
3 ---
4 > if (true) {
```

In this example, we have at least two mutation operators:

- `NonVoidMethodDeletion`: removes a call to a non-void method; in this case, the call to `match()` was deleted;
- `RemoveConditional`: removes conditional expressions with either *true* or *false*; in this case, the expression `inflection.match(word)` got replaced by `true`.

The two operators overlap and, thus, we need to take special care when analyzing the following transformations to the AST:

As we can see, there is:

- a **delete operation** of the method invocation node inside the `if` condition, which deletes its children;
- an **insert operation** of the literal value `true` in the place where the previous deletion occurred.

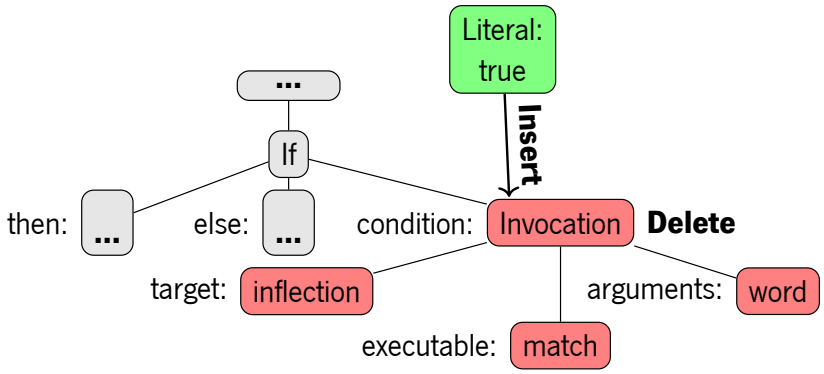


Figure 3.3: Two overlapping mutation operators

As mentioned before, two operators overlap:

- `NonVoidMethodDeletion` is identified by the delete operation, for which we check if the deleted node corresponds to an *Invocation* node and if the return type is not void (*boolean* in this case);
- `RemoveConditional` is identified by both operations - delete and insert. For this, we need to check if the deleted node's parent is of *If* type, if the inserted one is a *Literal* node and its value is a *boolean* (here, *true*) and if the inserted node is placed in the same spot as the previously deleted node, i.e., if both nodes' parent (*If*) is the same.

Independent mutations

Sometimes, the different places in a program where mutations are applied are independent of one another. Similarly, analyzing fragments of the complete set of AST modifications helps in detecting all the various operators.

```
191c191
< int oldCapacity = oldTable.length;
---
> int oldCapacity = oldTable.length-1;
260c260
< next = n;
---
> next = k;
303c303
< if (x == y) {
---
> if (x != y) {
```

In this example, three mutation operators were applied in entirely different places of the

program:

- `ArithmeticOperatorInsertion`: an arithmetic operator (`+, -, *, /, %`) is inserted, performing an operation between an existing variable/constant in the code and an inserted operand; here, the expression `oldTable.length` became `oldTable.length - 1` (Figure 3.4);
- `VarToVarReplacement`: a variable's name is replaced by another one; here, variable `k` replaces `n` (Figure 3.5);
- `RelationalOperatorReplacement`: a relational operator (`>, >=, <, <=, ==, !=`) is replaced by a different one; here, == was replaced by != (Figure 3.6).

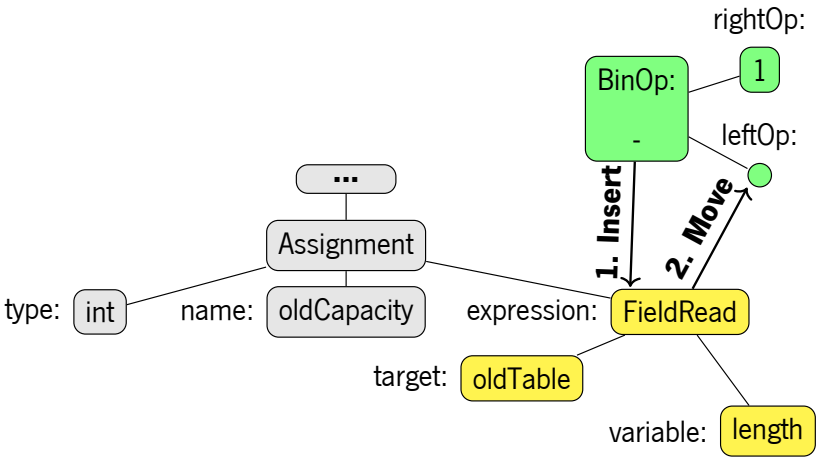


Figure 3.4: Arithmetic operator insertion

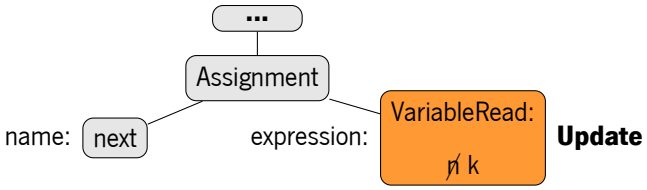


Figure 3.5: Variable replacement

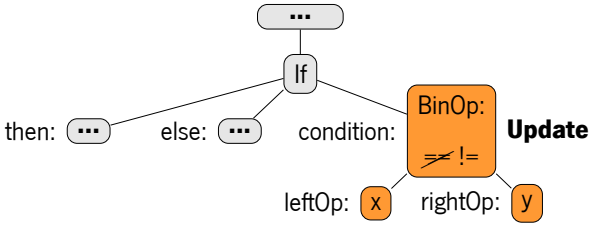


Figure 3.6: Relational operator replacement

All three previous images illustrate the modified parts of the complete AST. For each mutation operator, we need to check for different properties/rules:

- `ArithmeticOperatorInsertion`: check if the inserted node is a *BinaryOperator* and if its type is arithmetic. We also need to verify if the inserted node is taking the spot of the moved node. Additionally, it is necessary to confirm that neither of the operands is a *String*, as the binary operator + is also used for string concatenation;
- `VarToVarReplacement`: check the update operation is being performed on a *VariableRead*;
- `RelationalOperatorReplacement`: check if the update operation is performed on a *BinaryOperator* node and if it is of *relational* kind.

3.4 Inference Technique and Tool

This section describes the technique's algorithm and its implementation in the *Morpheus* tool.

3.4.1 Algorithm

The algorithm for inferring mutation operators from AST transformations can be divided into two phases (Algorithm 1):

1. **partitioning —** subdivides the set of AST operations so that each resulting subpart can be analyzed separately, isolating it from the "noise" of all other transformations;
2. **matching —** the subsets from the previous phase are matched against the mutation operator patterns. An analogy can be made regarding *type inference* systems, where expressions are associated with their correct types via automatic inference. Here, a set of transformations are "expressions" for which we are trying to associate with mutation operators ("types"). As we do not have any specific information regarding the nature of the mutation (which would correspond to type annotations in languages using *manifest typing*), we have to detect specific properties (*type rules*) in the provided transformations to conclude what mutation operator it represents.

To better illustrate the first part of the algorithm, let us consider a list of the form $[\texttt{Operation}_{node}]$:

```
1 [Delete_A, Insert_B, Move_C]
```

```
Data: An AST Diff between trees T1 and T2 containing the list of modifications M which, if applied to
      T1, will produce T2; a list of mutation operators MO to search for; an empty list of inferred
      mutation operators IMO
Result: The list of inferred mutation operators IMO
subLists ← [];
for i ← 1,size(M) do
    add([Mi], subLists); // appends list to subLists
    for j ← i+1,size(M) do
        add([Mi, Mj], subLists);
        for k ← j+1,size(M) do
            add([Mi, Mj, Mk], subLists);
        end
    end
end
foreach list ∈ subLists do
    foreach mo ∈ MO do
        inferred ← matches(mo, list);
        if inferred != null then
            add(inferred, IMO);
        end
    end
end
```

Algorithm 1: Algorithm for inferring mutation operators

This example indicates that we should apply one *delete*, one *insert* and one *move* operation to the original program tree to obtain the target program tree. Applying the previously mentioned sub-listing procedure would result in the sub-lists:

```
1 [ [Delete_A], [Delete_A, Insert_B],
2   [Delete_A, Insert_B, Move_C], [Delete_A, Move_C],
3   [Insert_B], [Insert_B, Move_C], [Move_C] ]
```

The first part of the algorithm is responsible for partitioning the complete list of AST modifications into sub-lists, each with a size between one and three (as these correspond to the minimum and the maximum number of changes that define a mutation operator, respectively). There are two reasons for this. First, because a program may have been modified in a way that reflects the application of more than one mutation operator, we need to identify multiple patterns in the complete list of alterations. As such, considering smaller portions of modifications enables us to find such patterns. Second, although the changes made to the AST are provided in *sequential* order, it does not mean that the ones characterizing the mutation operators are *contiguous*. The described approach allows us to consider *non-adjacent* sets of modifications in the search space.

The *matching* phase of the algorithm looks for predefined patterns in the list of changes to the AST. In such AST, different expressions of the language are represented in different subtrees.

The subtrees are constructed according to the productions/constructors defining the respective expression. In languages that use *type inference* systems, the inference mechanism traverses these trees and takes a different approach according to the type of node it is visiting. In other words, the node's type influences what rules are checked for the system to try to infer the correct data type.

Let us consider a simple syntax for expressions. Here, expressions can be a number, a variable's name or a binary operator that allows for more sub-expressions.

```
1 Expression = Num n | VarName n | BinOp
2 BinOp = Expression Op Expression
3 Op = + | - | / | *
```

A type inference system should have a mechanism that will take one of the possible expressions and, depending on the type it has in the tree, carry on with the appropriate action to determine the data type.

```
1 infer(expr: Expression): Type
2   switch(expr)
3     Num n -> //check if n is Int, Float, ...
4     VarName n -> //check scope for variable n
5     BinOp e1 op e2 ->
6       infer(e1);
7       infer(e2);
```

As we can see by the pseudo-code of a hypothetical `infer` function, the `switch` statement will perform different checks for different node types. That is, each node type encompasses its own set of rules.

The algorithm we describe takes a similar approach, as we can see in line 1. Each mutation operator comprises the group of rules it will analyze to report if some series of transformations complies with them. Let us take the example of the `ConstantReplacement` mutation operator, which changes the value of a constant in the source code. One of the inference rules for this case is expressed like:

M = program modification with set of transformations **T** = $\mathbf{T_1}$, ..., $\mathbf{T_k}$

$\mathbf{orig_k}$ = original node of $\mathbf{T_k}$ $\quad$ $\mathbf{new_k}$ = new node created by $\mathbf{T_k}$

$\mathbf{op_x}$ = operand of node **x** $\quad$ $\mathbf{p_x}$ = parent node of element **x**

$$\frac{\Gamma \vdash size(T) = 1 \qquad \Gamma \vdash T_1 : Update \qquad \Gamma \vdash orig_1 : Lit \qquad \Gamma \vdash new_1 : Lit}{\Gamma \vdash M : ConstantReplacement}$$

In this case, the first thing to do is to check the number of transformations to the tree. If T only has one transformation (T_1), this needs to be an *Update* operation and both the original node and the modified one have to be of type *Literal* (representing constants) to confirm the mutator. An example would be changing `methodCall(1)` to `methodCall(2)`.

This `ConstantReplacement` mutator can also be present through another pattern consisting of two transformations, T_1 and T_2. If so, we analyze the case where the constant value changed signals, e.g. from `methodCall(0)` to `methodCall(-1)`.

The inference rule for this situation can be expressed as:

$$\frac{\begin{array}{c}\Gamma \vdash size(T) = 2 \qquad \Gamma \vdash T_1 : Delete \qquad \Gamma \vdash T_2 : Insert \\ \Gamma \vdash orig_1 : Lit \qquad \Gamma \vdash orig_2 : UnaryOp \qquad \Gamma \vdash op_{orig_2} : Lit \qquad \Gamma \vdash p_{orig_1} = p_{T_2}\end{array}}{\Gamma \vdash M : ConstantReplacement}$$

For this example, operations T_1 and T_2 need to be a *Delete* and an *Insert*, respectively. The *Delete* operation corresponds to removing the constant 0 from the code. As such, the node to which the deletion operation is applied, $orig_1$, needs to be of type *Literal*. Because the new value for the constant is -1, we have to consider this as the addition of two separate elements: a *unary operator* representing the *negative signal* and a *literal* representing the number 1. Following this line of thought, the insertion operation T_2 needs to be applied to a *UnaryOp* node, $orig_2$. Furthermore, we also need to check if the operand associated with the unary operator is a constant, that is, a *Literal* node. Note that, although two nodes are inserted, we only consider the insertion of the top-level one, the *UnaryOp*, as the *Literal* node corresponding to the value -1 is its child. Also, we need to check if this deletion and insertion occurred in the same spot in the tree, which means the parent node of the deleted one must be the same as the parent of

the insertion operation, represented by the expression $p_{orig_1} = p_{T_2}$. Figure 3.7 illustrates these modifications.

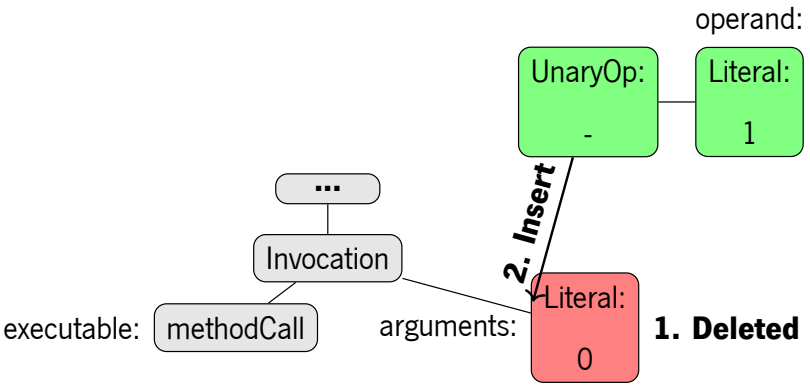


Figure 3.7: Constant replacement - positive to negative value

3.4.2 Morpheus

We have implemented our technique as the *Morpheus* tool[3]. *Morpheus* analyzes *Java* programs and was developed using the *Kotlin* language. As shown in Figure 4.1, it consists of two components: The *Diff Calculation* gets as input the original and the mutated programs and produces the list of transformations representing the differences between the programs (Falleri et al., 2014). The second component - the *Inferrer* - implements Algorithm 1, with its two parts: *partitioning* and *matching*.

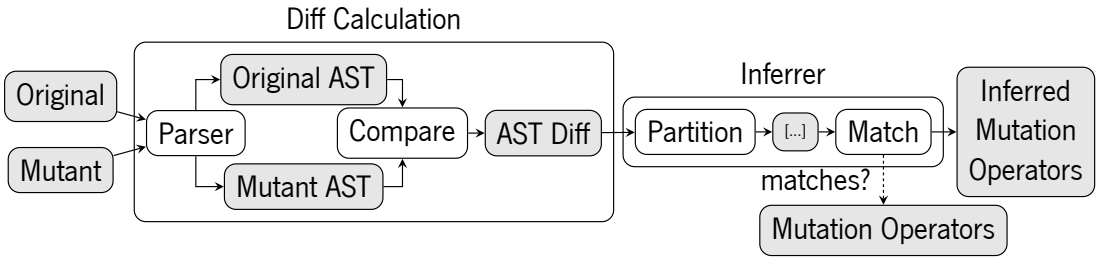


Figure 3.8: *Morpheus* architecture

Morpheus is an extensible tool: it implements all operators in Table 3.1 and can easily be extended with new mutation operators due to its extensible architecture.

[3] `https://github.com/FranciscoRibeiro/morpheus`

3.5 Dataset and Analysis

In this section, we present the structure of the produced dataset and analyze the results. *Morpheus* was used to analyze the *CodeDefenders* (Rojas et al., 2017) repository[4] containing 20 original programs and 1753 mutants created by players. In *CodeDefenders*, attackers modify programs to introduce faults and defenders write unit tests that detect these errors. The 20 original classes are part of various real-world open-source projects. We identified 230 mutants that failed to compile and 27 for which the *AST diff* tool failed to produce an edit script, leading to 1496 valid programs.

3.5.1 Dataset Structure

After using our tool to analyze every valid mutant in the repository, we produced a dataset[5] containing information about each one. Therefore, each mutant has a corresponding record in the dataset with the following fields:

- **Mutant ID:** Mutant identifier based on the repository;
- **Nr. AST modifications:** Number of modifications applied to the original AST;
- **AST modifications:** List with the types of operations performed on the AST in order to obtain this mutant;
- **Mutation overviews:** List of the source code change for each inferred mutation operator;
- **Inferred mutation operators:** List containing the names of the inferred mutation operators (according to Table 3.1);
- **Callables:** List containing the method/constructor names where each mutation was inferred;
- **Old start-end lines:** List containing the start and end lines in the *original* file where each mutation operator was detected (same index in inferred mutation operator list);
- **Old start-end columns:** Same as the previous field but for columns;
- **New start-end lines:** List containing the start and end lines in the *mutated* file where each mutation operator was detected;
- **Start-end columns:** Same as the previous field but for columns;
- **Relative old start-end lines:** List containing the start and end lines inside the callable's

[4] `https://study.code-defenders.org/`
[5] `https://doi.org/10.6084/m9.figshare.15173934`

body in the *original* file where each mutation operator was detected;

- **Relative new start-end lines:** Same as the previous field but for the *mutated* file.

Recalling the example in Figure 3.2, the fields for the corresponding dataset record have the values shown in Table 3.2.

Table 3.2: Dataset record for one of the mutants

Mutant ID	`1001/15/00000001/ByteArrayHashMap`
Nr AST Modifications	2
AST Modifications	`[DeleteOperation, MoveOperation]`
Mutator Overviews	`[AOD(from=(length - 1),to=length)]`
Inferred Mutation Operators	`[ArithmeticOperatorDeletion]`
Callables	`[ByteArrayHashMap#indexFor(int,int)]`
Old Lines	`[324-324]`
Old Columns	`[16-27]`
New Lines	`[324-324]`
New Columns	`[14-21]`
Relative Old Lines	`[1-1]`
Relative New Lines	`[1-1]`

3.5.2 Dataset Analysis

Table 3.3 shows the number of mutants associated with each class in the repository. Furthermore, it also displays the number of mutants for which *Morpheus* could infer and classify as corresponding to one or more mutation operators.

The effectiveness rate of our technique is calculated in terms of the number of mutants for which we can infer at least one mutation operator divided by the total number of valid mutants. Overall, we were able to infer 78% of all the considered mutants. This percentage is not consistent throughout all of the programs. However, there is not a single program for which we could not detect the presence of a mutation operator. The class with the least amount of inferred mutation operators was `VCardBean`, with 67% of its mutants classified. The `XMLParser` class is on the opposite side with all of its mutants inferred and, therefore, a 100% effectiveness rate. Curiously, the same mutation operator was applied in the same manner by all the players who had to attack this class, i.e., create mutants. This particular mutation occurred in a method that replaces occurrences of the character "&lt" with the character "<". The arguments of the call to the method in question were passed as `string` literals. As such, every call to `replaceAll("<", "<")`

Table 3.3: Available mutants per valid program

Class	#Available Mutants	#Inferred Mutants	Effectiveness %
ByteArrayHashMap	126	108	86%
ByteVector	55	39	71%
ChunkedLongArray	95	68	72%
FontInfo	30	22	73%
FTPFile	34	24	71%
HierarchyPropertyParser	66	48	73%
HSLColor	50	42	84%
ImprovedStreamTokenizer	84	70	83%
ImprovedTokenizer	130	97	75%
Inflection	13	10	77%
IntHashMap	71	55	72%
ParameterParser	68	57	84%
Range	152	122	80%
RationalNumber	47	40	85%
SubjectParser	28	21	75%
TimeStamp	32	25	78%
VCardBean	173	116	67%
WeakHashtable	40	32	80%
XmlElement	175	136	78%
XMLParser	27	27	100%
Total	**1496**	**1159**	**78%**

was mutated to `replaceAll("<", "<")`. As string literals are considered constants, all these mutations were inferred to be the `ConstantReplacement` operator.

We did not infer any mutation operators for 337 of the mutants — Table 3.4. Ideally, a mutation is a slight syntactic modification that alters the program's behavior. However, sometimes, the generated mutants are not simple modifications because they consist of extensive edits to the source code. These changes are challenging to infer as no mutation operator resembles it. On the other hand, some of these mutants are still small. Nevertheless, they represent intricate code modifications. The following example illustrates such a situation.

```
1 103c103
2 < set(index2, tmp);
3 ---
4 > set(index2, get(index2-1));
```

Here, a variable `tmp` got replaced by a method call with different arguments. Many changes are co-occurring, making it difficult to discern the logic behind them.

Table 3.4: Inferences per mutation operator

Mutation Operator	#Occurrences
ConstantReplacement	273
RelationalOperatorReplacement	179
VarToVarReplacement	141
ArithmeticOperatorInsertion	105
StatementDeletion	104
NonVoidMethodDeletion	90
VarToConsReplacement	83
ReturnValue	54
UnaryOperatorInsertion	47
ConditionalOperatorReplacement	42
VoidMethodDeletion	40
ArithmeticOperatorReplacement	28
AccessorModifierChange	21
UnaryOperatorReplacement	20
RemoveConditional	20
ConditionalOperatorDeletion	18
ArithmeticOperatorDeletion	16
ConsToVarReplacement	13
MemberVariableAssignmentDeletion	10
ConditionalOperatorInsertion	9
ConstructorCallReplacementNull	7
AccessorMethodChange	6
UnaryOperatorDeletion	5
StaticModifierDeletion	4
ReferenceReplacementContent	4
TrueReturn	4
ArgumentNumberChange	4
BitshiftOperatorReplacement	4
FalseReturn	2
StaticModifierInsertion	2
ArgumentTypeChange	2
BitwiseOperatorReplacement	1
BitshiftOperatorDeletion	1
Negation	1
UNCLASSIFIED	337

To get an overview of the entire set of unclassified mutants, we compared every program version for which *Morpheus* did not produce any inference against its original and verified that the difference did not match the criteria of any mutation operator. Nevertheless, we still detected some recurring patterns, shown in Table 3.5. The most frequently detected pattern is adding an instance method call to a variable, which happened 51 times. Adding a statement

was the second most spotted type of mutation with 45 occurrences. This is the least specific type of transformation and one of the most difficult to incorporate in mutation testing, as deciding which source code to add to a specific part of a program is not a straightforward task. The third most common mutation pattern was replacing the kind of exception thrown, occurring 21 times. Curiously, we can observe that some patterns displayed in Table 3.5 parallel with some of the mutation operators covered by *Morpheus*. For instance, let us consider the fifth most common one, *Overwrite Default Initialization*, which assigns a specific value to a variable, thus not allowing the default ones to occur. This pattern can be seen as the transformation opposing the *MemberVariableAssignmentDeletion* operator (Table 3.1), which eliminates specific assignments to member variables. As another example, if we consider the expression $x < 2$ and then apply the *Negate Expression* pattern, we would get $!(x < 2)$. From another perspective, this is a particular case of rewriting this expression as $x >= 2$, which ends up being covered by the operator *RelationalOperatorReplacement*. The detection of these new patterns has implications regarding mutation-based repair techniques (Debroy and Wong, 2010; Martinez and Monperrus, 2016; Durieux et al., 2019; Rothenberg and Grumberg, 2016; Debroy and Wong, 2014; Repinski et al., 2012). The candidate fixes for a faulty program are produced by applying mutation operators to suspicious parts of the source code. As such, a repair technique of that kind would not generate an appropriate patch for the cases from which we extracted the patterns reported in Table 3.1. This is because these particular faults originated from applying changes that are not covered by any mutation operator in the literature to the best of our knowledge. Even though the patterns we found in the unclassified cases are used to introduce faults, instead of producing fixes, it is still essential that repair tools incorporate these new operators. As we stated before, some of them revert the effects of already documented operators (*Overwrite Default Initialization* reverts *MemberVariableAssignmentDeletion*). Moreover, the most challenging patches to create are the ones that require adding code (Debroy and Wong, 2010), which patterns like *Instance Method Call Addition* and *Add String Concat* aim to achieve.

As discussed in Section 3.3.1, sometimes, a mutant can consist of several mutation operators, also called *higher-order mutants* (Jia and Harman, 2009), and *Morpheus* can detect these occurrences. Table 3.6 shows the frequency of the number of mutation operators for each program alternative. As we can see by the table, the most common mutants are the ones that get only one mutation operator inferred, totaling 1004 mutants. The program versions with more

Table 3.5: Manual inspection — detected patterns in the 337 program versions with no reported inferences

Pattern	Example	#Occurrences
Instance method call addition	var → var**.method()**	51
Add statement		45
Throw exception replacement	throw new ~~A()~~ **B()**	21
Replace with new instance	x = ~~var~~ **new Var()**	20
Overwrite default initialization	int x; → int x **= 4**;	20
Replace Method Call	var.foo(); → var.**bar()**;	15
Final keyword removal	~~final~~ int x;	14
If block deletion	~~if(cond) x = 2;~~	11
Return type change	public ~~int~~ **long** foo()	9
Delete statement		8
If check deletion	~~if(cond)~~ x = 2;	7
Add string concat	str → str **+ ”word”**	7
Continue/break replacement	~~continue~~ **break**;	7
Swap lines		7
Primitive to wrapper	~~int~~ **Integer** x = 2;	7
Change thrown exception for return	~~throw new A()~~ **return -1**	6
Instance change	var1.foo() → **var2**.foo()	6
Negate expression	if(expr) → if(**!**expr)	5
Delete case	~~case SOME_VALUE:~~	5
Change increment size	i++ → i+=2	4
Change assigned	x = 2 → **y** = 2	3
Delete string concat	str ~~+ ”word”~~	2
Equivalent default initialization	Obj x ~~= null~~; → Obj x;	2
While/If Replace	while(cond) → **if**(cond)	1
Variable Type Change	int var; → **long** var;	1
Delete Try/Catch		1
Change Constant Type	Integer.MAX → **Long**.MAX	1
Add Else Block		1
Undefined		50

Table 3.6: Mutation operators per mutant

#Mutation Operators	1	2	3	4	5	16
#Files	1004	132	14	6	2	1

than one inferred mutation operator combine for 155, representing 10% of all the valid mutants in the repository.

3.6 Mutation-based Repair

We can devise a repair strategy that takes advantage of this new information by translating the bug-inducing changes in terms of mutation operators. Our implementation[6] of such a repair strategy is divided into three parts:

- Extracting fault localization components: interprets the report produced by *Morpheus* and creates components that connect the inferred mutations to their location in the source code;
- Finding nodes in the AST: isolates tree nodes representing source code elements in specific locations;
- Reverting mutations: applies the opposed mutation operator to produce patches.

Figure 3.9 shows how these parts connect.

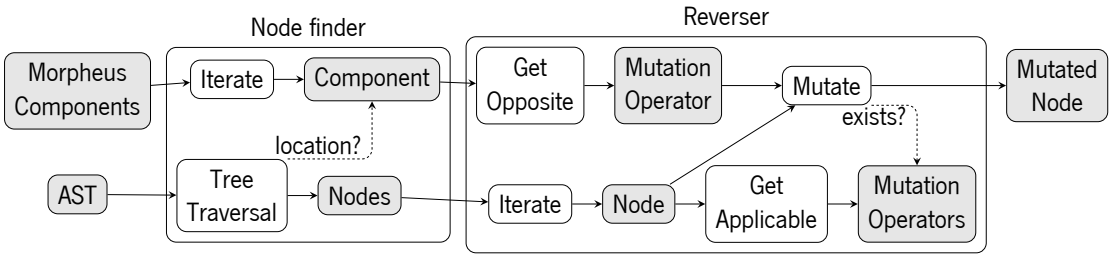


Figure 3.9: Repair overview

3.6.1 Extracting mutation operators' locations

We can create components that allow us to kickstart the repair process by associating each inferred mutation to its location in the source code. Let us go back to the introductory example. The corresponding component would convey the information in Table 3.7.

It shows that the *argument number change* mutation operator was inferred in line 133 and spans columns 61 to 74. Furthermore, the transformation was detected in the 8th line of the `getEnumProperty` method. These components can be extracted from the output provided by *Morpheus*.

[6] https://github.com/FranciscoRibeiro/auto_repairer

Table 3.7: Mutation's location: introductory example

Mutation Operator	`ArgumentNumberChange`
Callable	`getEnumProperty(Class,String)`
Start-End Old Lines	132-132
Start-End Old Columns	40-61
Start-End New Lines	133-133
Start-End New Columns	61-74
Start-End Old Relative Lines	8-8
Start-End New Relative Lines	8-8

3.6.2 Finding AST nodes

Since these components can pinpoint specific places in the buggy source code, the repair strategy can then analyze the program's AST to find the corresponding nodes.

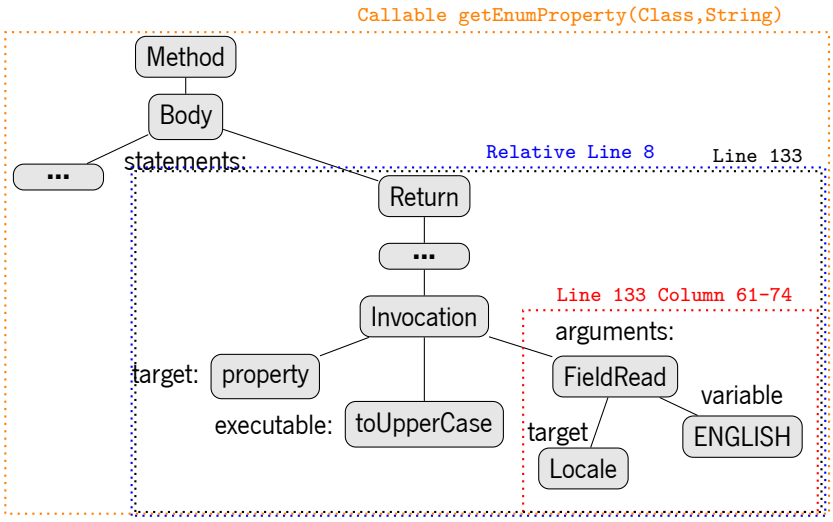


Figure 3.10: Finding AST nodes: introductory example

Figure 3.10 illustrates four different criteria to find nodes in an AST:

1. matching both lines and columns;
2. matching only lines;
3. matching relative lines;
4. matching the callable;

As Figure 3.10 shows, a different number of nodes may be retrieved depending on the selected criteria. As such, it is a matter of deciding on efficiency (1) *vs.* efficacy (2). To increase the likelihood of generating a fix, we should seek to detect many nodes, albeit at the expense of producing a large number of patches. On the other hand, we can limit the number of generated

patches by only fetching nodes matching both the lines and columns reported by *Morpheus*.

However, the detected mutations are not always in the same file that we wish to repair. Such would occur when *Morpheus* infers mutations in past file versions, but we wish to use that information and repair a more recent version of a program. In these cases, the reported line numbers may differ from the current location. Thus, it is helpful to use the reported information about the relative line numbers (3) or the callable (4), representing efficiency and efficacy, respectively.

3.6.3 Reversing mutations

Following the previous step, the repair process iterates over the returned AST nodes and tries to mutate each one — Algorithm 2. Every mutation operator that *Morpheus* can infer has another one that performs the opposing transformation. As such, the opposing mutation operator is considered for the inferred mutation in a component (line 2). Then, the strategy retrieves the appropriate mutation operators regarding the node type in question (line 2). Finally, if the opposing transformation belongs to the group of applicable mutations (line 2), it applies it over the AST node (line 2).

Data: The inferred mutation operator IMO; an AST node N that we wish to mutate; a mapping of opposing mutation operators $MapO$; a mapping between types of AST nodes and their applicable mutation operators $MapN$; an empty list of mutated nodes $MutN$
Result: The list of mutated nodes $MutN$
opposite ← $MapO[IMO]$; // `get opposite mutOp of` IMO
mut_ops ← $MapN[N]$; // `get mutOps applicable to` N
if *opposite* ∈ *mut_ops* **then**
  $MutN$ ← repair(opposite, N); // `apply opposite to node` N
end

Algorithm 2: Algorithm for reversing mutation operators

3.7 Case Studies with Real Bugs

The main idea we want to deliver is that the *semantics* behind a *bug* can guide the repair process of a program. To show this, we used *Morpheus* to analyze *Bugswarm* and *Defects4J*, extracting several case studies from real-world programs. Furthermore, we also implemented an automated repair process that successfully fixed all the studied programs. These experiments are available

for replication[7,8].

Bugswarm We selected case studies from this repository for which a simple fix, susceptible of being represented by a mutation, makes the program pass every test. We looked back through the commit history of each selected case study and searched for the most recent commit that began failing these tests. There, we detected mutations opposing the bug-fixing modification, likely responsible for introducing the bug.

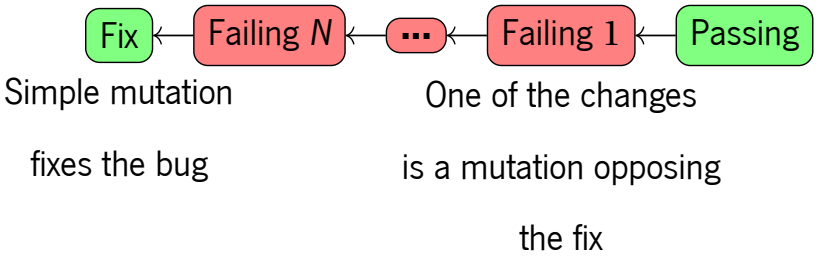


Figure 3.11: Selection criteria for the case studies - *Bugswarm*

Defects4J In this repository, the fixes for the selected case studies can also be obtained by performing a simple mutation. In *Defects4J*, however, the commit in which we detect the opposing transformation does not necessarily cause the software to fail its tests. Moreover, the file that makes the tests fail may have changed multiple times since introducing the original reason for the bug.

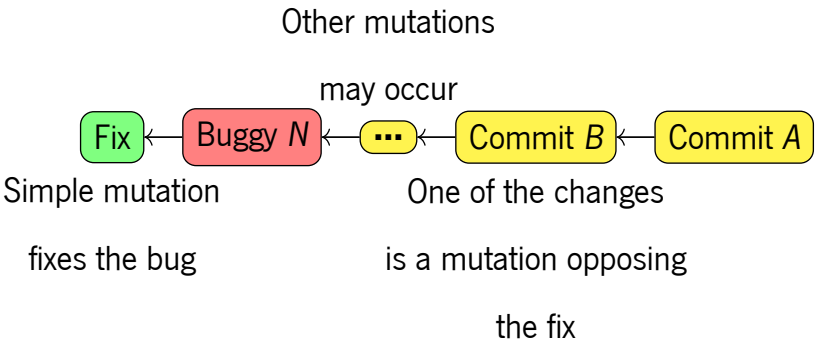


Figure 3.12: Selection criteria for the case studies - *Defects4J*

Besides repairing all analyzed programs, when guiding the repair process, our approach has four key advantages over spectrum-based fault localization (SFL) reports:

[7] `https://github.com/FranciscoRibeiro/bugswarm-case-studies`
[8] `https://github.com/FranciscoRibeiro/d4j-case-studies`

1. **Efficiency:** *Morpheus* detects fewer mutants when compared against the many lines SFL reports. Moreover, SFL sometimes misses to rank the faulty line in which *Morpheus* can spot a mutant;
2. **Compilation:** buggy programs may fail to compile as *Morpheus* does not need the program's execution trace, whereas SFL would not generate a report;
3. **Reachability:** mutations can be detected in any commit of a program's history, regardless of whether there are failing tests or not;
4. **Granularity:** inferred mutants have more granularity than the line number provided by SFL, allowing program repair to use different criteria (recall Figure 3.10).

The interested reader can find a detailed explanation of the case studies on the page:

`github.com/FranciscoRibeiro/qrs21-case-studies-report`

3.8 Threats to Validity

Our work has two main objectives. Firstly, to assess whether we can translate the evolution of a program in terms of mutation operators. More precisely, are the changes applied to a program equivalent to the application of well-known mutation operators? Secondly, to check if the information about inferred mutations can benefit automated program repair. That is, can programs be fixed more efficiently by using this new knowledge over regular SFL reports?

As we have shown, the answer to the previous questions is yes. There are, however, several aspects that may affect the validity of our work.

Internal Validity There is no guarantee that the analyzed programs compare equally in terms of susceptibility to mutations. Mutations are slight syntactic modifications that alter the program's semantics, and, as such, the examined mutants may consist of more complex changes which do not correspond to any documented mutation operators. Furthermore, the mutants from *CodeDefenders* were created by people learning about the topic in question. Although some mutants may not convey the desired simple nature, we think our results show that a considerable part of them do hold to this standard.

External Validity The mutants from the *CodeDefenders* repository were created with the explicit goal of producing faults. Nonetheless, we showed that our work generalizes to a real-world context by using the inferred information to repair open-source projects from the *Bugswarm* and the *Defects4J* repositories in which faults were unintentionally introduced. Quoting DeMillo et al. (1978) on a quality about programmers: "they create programs that are close to being correct!".

Construct Validity The *CodeDefenders* mutants were produced by subjects who were aware that they were to be used in research. We do not believe this to have compromised our measures because the origin of the repository is independent of our study, and the intentions of inferring mutation operators were never communicated. Moreover, our analysis of real-world faults further strengthens this point as programs were developed in a completely disconnected context disassociated from any research intentions.

Conclusion Validity The idea transmitted to *CodeDefenders* players was they should replicate the behavior of mutation operators. Still, some mutants did not obey this practice. We conclude that real-world program changes can be described in terms of mutation operators, as demonstrated by the reported real-world case studies. Furthermore, previous studies have already shown associations between real faults and mutation operators.

3.9 Related Work

Jia and Harman (2009) present the concept of *higher-order mutation testing*, in which mutants are not individual faults but are composed of several faults. They emphasize *subsuming higher-order mutants*, which are notably hard to kill. The program versions for which *Morpheus* infers two or more mutation operators are instances of higher-order mutants.

Debugging is one of the most expensive actions in the development cycle (Vessey, 1985) and a considerable effort is put into fault localization (Parnin and Orso, 2011; Ang et al., 2017). *MUSE* (Moon et al., 2014) applies mutations to both faulty and correct statements to rank the most suspicious lines, improving over previous state of the art. The reasoning is that tests that pass in the original program are more likely to fail when correct statements are mutated and less likely to do so when faulty lines are mutated. Mutation-based fault localization has also been applied

to projects written in multiple programming languages, reporting high accuracy and proposing new mutation operators (Hong et al., 2015). Other approaches (Papadakis and Le Traon, 2012, 2015) detect suspicious statements by calculating similarities between mutants. Zhang et al. (2013) describe a technique in which artificially produced mutants are mapped to higher-level programming edits. Instead, we provide more granularity by interpreting structural changes to infer what mutation operator is being applied. Fault localization lacks the semantics behind the faults it spots. *Morpheus* can provide this as it computes the context of the modifications it detects.

Mutation-based program repair (Debroy and Wong, 2010; Martinez and Monperrus, 2016; Durieux et al., 2019; Rothenberg and Grumberg, 2016; Debroy and Wong, 2014; Repinski et al., 2012) uses fault localization to mutate the most suspicious lines. A mutant is considered as a potential fix if it passes all the test cases. If effectiveness is the focus, a large set of mutation operators should be considered to cover the largest number of faults. On the other hand, techniques aiming for efficiency should only apply a small set of mutation operators to minimize overhead, sacrificing the ability to fix some types of faults. *Morpheus* infers mutation operators and repair strategies can take advantage of this semantics to apply modifications that revert the faulty effects. Tan and Roychoudhury (2015) aim to repair regression errors by manually extracting fix patterns from a project's history and applying them to suspicious statements. Our approach differs, as the inference process of our tool automatically detects the application of mutation operators. Moreover, we aim to infer operators used by mutation testing tools, instead of high-level transformations such as "Revert to previous statement". The automatic detection of bug fixes (Madeiral et al., 2018) is also based on an established taxonomy, with changes being analyzed at the *AST* level. However, the list of considered bug fixes — 25 patches — is more generic and not as extensive as ours — 34 mutations.

Generating test cases through mutations (Fraser and Zeller, 2010) has been applied in web page testing (Almeida et al., 2019), and tools like *Sapienz* (Mao et al., 2016) are already following this approach and successfully detecting bugs in mobile applications used by millions of people.

Tree differencing has been applied to build files (Macho et al., 2017). Hence it is well suited to address a project's configuration. Our work focuses on a much broader aspect, requiring the *ASTs* of source code to reason about the issues.

Some approaches (Hanam et al., 2016) mine project repositories to find frequent bug pat-

terns for languages that research has yet to report mutation operators. Our work differs, as we present a tool that automatically detects the application of well-documented mutation operators to a correct program.

3.10 Summary

We presented an inference technique that defines the context behind source code changes by associating them with well-known mutation operators. We implemented it as the *Morpheus* tool and analyzed several manually modified programs. Our results show that this is a sound approach, as we were able to infer mutation operators for 78% of the 1496 valid mutants in *CodeDefenders* and that 10% of these are *higher-order mutants* (Jia and Harman, 2009). Furthermore, we have also analyzed several case studies extracted from real-world projects in *Bugswarm* and *Defects4J*, showing the benefits of our approach regarding automated program repair. We fixed these programs by implementing a repair tool that reverts bug-introducing changes based on Morpheus' information by applying the opposing mutation. The concept of *higher-order mutants* was essential, as highlighted by the case studies. The repair strategy focused on fixing the effect of a single atomic mutation to create a patch for a program that was modified in separate places by different mutation operators.

Replication Package

The necessary resources to replicate this study and the full set of results, are publicly available:

- **Mutant repository:** `study.code-defenders.org`
- **Morpheus:** `github.com/FranciscoRibeiro/morpheus`
- **Repair tool:** `github.com/FranciscoRibeiro/auto_repairer`
- **Dataset:** `doi.org/10.6084/m9.figshare.15173934`
- **Case studies:**
  - **Bugswarm:** `github.com/FranciscoRibeiro/bugswarm-case-studies`
  - **D4J:** `github.com/FranciscoRibeiro/d4j-case-studies`
  - **Report:** `github.com/FranciscoRibeiro/qrs21-case-studies-report`

Program Repair as Code Completion

4

"Framing Program Repair as Code Completion"
Francisco Ribeiro, Rui Abreu, and João Saraiva
In: Proceedings of the 3rd International Workshop on Automated Program Repair, APR 2022
Pittsburgh, USA, May 19, 2022
DOI: **10.1145/3524459.3527347**

Abstract *Many techniques have contributed to the advancement of automated program repair, such as: generate and validate approaches, constraint-based solvers and even neural machine translation. Simultaneously, artificial intelligence has allowed the creation of general-purpose pre-trained models that support several downstream tasks. In this chapter, we describe a technique that takes advantage of a generative model — CodeGPT — to automatically repair buggy programs by making use of its code completion capabilities. We also elaborate on where to perform code completion in a buggy line and how we circumvent the open-ended nature of code generation to appropriately fit the new code in the original program. Furthermore, we validate our approach on the ManySStuBs4J dataset containing real-world open-source projects and show that our tool is able to fix* 1739 *programs out of* 6415 *— a* 27% *repair rate. The repaired programs range from single-line changes to multiple line modifications. In fact, our technique is able to fix programs which were missing relatively complex expressions prior to being analyzed. In the end, we present case studies that showcase different scenarios our technique was able to handle.*

4.1 Introduction

Automated Program Repair (APR) is a prominent field of software engineering. The continuing increase in complexity and size of software systems urges the community to invest its efforts on developing techniques that automatically identify patches that are able to fix faults arising from

the implementation of new functionalities and code maintenance (Goues et al., 2019). These patches are generated based on the original buggy program and take advantage of the fact that the developers' efforts result in almost accurate programs or, as DeMillo et al. (1978) put it: "they create programs that are close to being correct!". Many approaches have been developed that successfully achieve this repair task. Early works (Le Goues et al., 2012b; Arcuri, 2011) utilize genetic programming by considering a buggy program as seed which is then continuously evolved at each generation by producing different programs through mutation and crossover. Other approaches (Nguyen et al., 2013; Xuan et al., 2017; Durieux and Monperrus, 2016) are constraint-based and analyze information from test executions to create constraints which are then fed to a solver to generate a patch.

More recently, APR techniques have taken advantage of machine learning advancements to build deep learning models. Some of these techniques (Chen et al., 2021; Ding et al., 2021; Li et al., 2020; Lutellier et al., 2020) employ Neural Machine Translation (NMT) to translate buggy code into fixed code. Likewise, general-purpose tools and models (Svyatkovskiy et al., 2020; Feng et al., 2020; Lu et al., 2021) supporting code understanding and code generation tasks have been developed.

We argue that the code generation capabilities of pre-trained models like CodeGPT can be leveraged to specifically target program repair, effectively treating it as a code completion task. Let us consider an example from a real-world open-source software project.

```
1 171c171
2 < ... keyValueSequence = new ArrayList<Data>(|);
3 ---
4 > ... keyValueSequence = new ArrayList<Data>(|entries.size());
5 271c271
6 < ... int mapLoadChunkSize = |nodeEngine.getGroupProperties().
      MAP_LOAD_CHUNK_SIZE.getInteger();
7 ---
8 > ... int mapLoadChunkSize = |getLoadBatchSize();
```

The previous code shows two buggy lines and their corresponding fixes underlined. The expressions that repair this program — `entries.size()` and `getLoadBatchSize()` — are not trivial to figure out, even if we know that lines 171 and 271 are responsible for this bug. More precisely, repairing this bug implies the developer not only determines the incorrect expressions but also how to expand them. However, the complexity of this task can be reduced to simply performing code completion on the spot highlighted by the vertical bar to replace the leading

code. We assume faulty line numbers are identified beforehand by well-established and accurate fault localization techniques (Jones et al., 2002; Pearson et al., 2017; Campos et al., 2012; Perez and Abreu, 2018; Wong et al., 2016).

We present a repair technique that, given a file and buggy line numbers, seeks to fix a program by computing the most appropriate columns to perform code completion and incorporating the generated code in the original program.

The contributions are:

1. a technique that repairs buggy programs based on code completion;
2. a publicly available implementation of such technique;
3. a validation on a dataset of real-world projects with results showing our technique is able to fix 1739 programs out of 6415, representing a 27% repair rate;
4. a case study investigation highlighting some capabilities of our work.

The original aim of code completion is to assist the developer while writing code. Throughout the report, we use annotations like the vertical bar representing the cursor position in a text editor and color highlights showing intended or generated completions. However, these serve to better visualize our approach's behavior. The primary goal of this work is to develop a technique and tool that uses code completion to produce patches without developer intervention.

4.2 Background

Research in Natural Language Processing (NLP) focuses on how natural language can be processed, analyzed and manipulated by computers. Although its roots are based on symbolic rules and statistical modeling, the more recent adoption of machine learning models has allowed this field to flourish as one of the most prevalent areas of study in computer science. The continued work by the community has led to the specialization of certain subtasks within NLP into well-defined processes. Natural Language Understanding (NLU) analyzes natural text and appropriately encodes it into more low-level representations, while Natural Language Generation (NLG) transforms these machine representations into natural language text. NLP has benefited immensely from the application of neural networks, which allowed for the development of complex but highly effective models like BERT (Devlin et al., 2019) and GPT (Radford et al., 2019) that

have achieved tremendous success in language understanding and language generation tasks, respectively. These state-of-the-art models are based on the Transformer (Vaswani et al., 2017) neural architecture, which has shown to be more advantageous than previously used RNN-based architectures for analyzing longer and deeply-rooted dependencies. This is, in most part, thanks to a self-attention mechanism which allows the model to create connections between every token in a sequence no matter the distance between them. As a consequence, the representation of each token will be affected by its relationship with other ones, creating a more meaningful aggregate representation.

More recently, inspired by the significant advances in this area, the community has also directed its focus to the application of NLP principles regarding programing languages. In fact, software developers have been incorporating these tools into their workflow as they find them to have a positive effect in their productivity. One of the most sought-after capabilities in these systems is code completion (Bruch et al., 2009) and every IDE or code editor supports this key feature. However, many of them provide this at a basic level, such as API call and parameter completion, limiting their usage to scenarios in which a developer needs to have a specific idea already typed in. Because of the success of pre-trained models like BERT and GPT, the architectures behind them have been used to create corresponding adaptations directly suited for programming languages. Thus, code understanding and code generation have allowed for advancements regarding the previous limitations through models such as IntelliCode Compose (Svyatkovskiy et al., 2020), CodeBERT (Feng et al., 2020) and CodeGPT (Lu et al., 2021).

CodeGPT is able to generate long and complex code sequences that are computed based on the context provided to the model. This input context consists of code preceding the point from which we wish the model to start generating more code. Essentially, the produced code sequence acts as a continuation of the original code piece. This way, CodeGPT can be used to perform code completion. In this work, we do not use this capability to help developers fill in the most suitable names for method calls or variable identifiers. Instead, we leverage it to inject new segments of code into existing buggy programs and modify their behavior. As a result, we show that we can take advantage of a code completion mechanism to conduct an entirely different task — automated program repair.

4.3 Repair Technique

Our approach can be divided in four components, as shown in Figure 4.1:

1. **Cutting:** the two inputs are the buggy file and the buggy line number. After the buggy file is parsed into its abstract syntax tree, we extract the nodes located at the buggy line number. As we mentioned, we assume the faulty line is already provided by some fault localization technique. Then, based on the criteria implemented in Algorithm 3 (described in Section 4.4), we compute the column numbers representing the places for which code completion is to be performed. Lastly, we truncate the buggy file at those columns, creating a file for each alternative;
2. **Code Generation:** we perform code completion for each truncated file by providing an array of tokens as context to *CodeGPT*. Through random sampling, the model generates several token sequences, thus producing alternative ways of continuing the input code. These sequences are decoded and output as strings.
3. **Bounding:** the code completion step is open-ended. That is, the model generates code without necessarily stopping at some suitable character regarding the language's syntax. As such, it is very likely that the last generated token does not terminate a well-formed expression or statement, as the output will finish once the maximum context size is reached. Likewise, the essence of the generated code may be sound except for the initial tokens. For this reason, we limit the generated code sequences based on relevant characters regarding the language's syntax to extract valid completions.
4. **Character Synchronization:** the final step is to attach the generated completions to the original buggy code. This is done by using characters that allow each completion to fall into place in the original buggy line, combining both pieces of code to produce a potential patch.

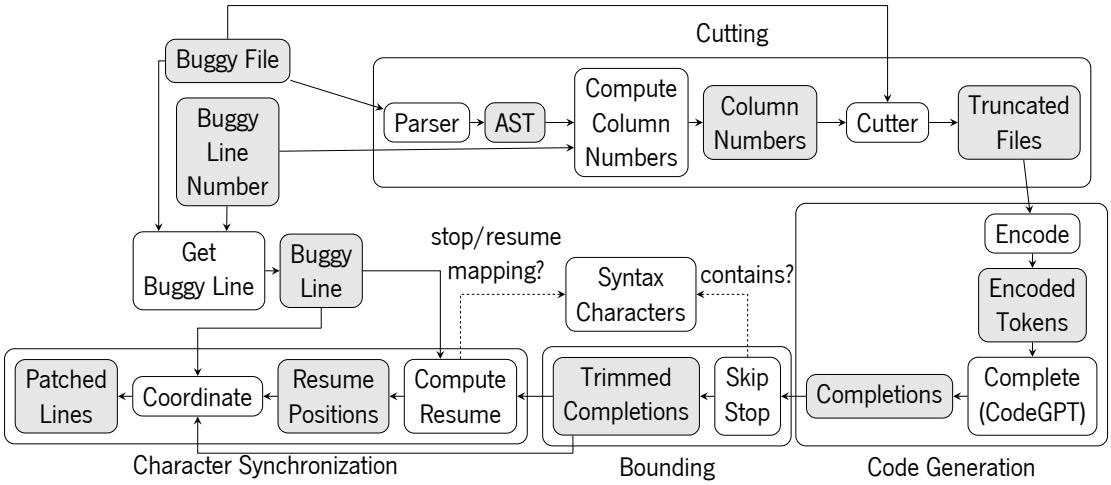


Figure 4.1: The four components of the system's architecture

4.4 Truncation Algorithm

As we have discussed before, in this work we leverage code generation by using the *CodeGPT* model to produce potential patches. We can interpret this code generation step as code completion being performed at a specific column in a line of code — similar to what is normally seen in text editors and IDEs.

Let us consider the the introductory example again. Listing 4.1 represents the desired fixed lines.

```
1 > ... keyValueSequence = new ArrayList<Data>(entries.size());
2 > ... int mapLoadChunkSize = getLoadBatchSize();
```

Listing 4.1: Introductory example — desired completion

For this case, code completion would happen at the illustrated cursor position and the code to be generated is highlighted in grey. Therefore, we need to compute the places in that line for which we want to perform code completion. As such, we devised an algorithm that aims to compute suitable column numbers for the purpose of generating code sequences. We implemented it as a tool and make it available[1].

We targeted two scenarios when designing the algorithm. Code completion is frequently useful when developers want to predict:

- Code to continue specific language constructs (e.g. methods to invoke after "."");

[1] https://github.com/FranciscoRibeiro/code-truncater

- Candidate names for partially written identifiers (e.g. writing a variable's name halfway through).

As such, our algorithm computes column numbers based on:

- Textual boundaries of language constructs represented by AST nodes;
- Camel-case and underscore separation of words according to Java naming conventions.

Data: An AST $T1$
Result: The list of computed column numbers $ColNrs$
ColNrs ← [];
foreach *node* ∈ $T1$ **do**
  start ← node.firstColNr; end ← node.lastColNr;
  add(start, ColNrs);
  add(end, ColNrs);
  if *node is Identifier* **then**
    name ← node.name;
    for *i ← 1,size(name)* **do**
      if *(i ≠ size(name)* ***and*** $name_i$ *is lowerCase* ***and*** $name_{i+1}$ *is upperCase)* **then**
        add(start+i+1, ColNrs);
      else if $name_i$ == '_' **then**
        add(start+i, ColNrs); add(start+i+1, ColNrs);
      end
    end
  end
end

Algorithm 3: Algorithm for computing column numbers

This means that our algorithm would truncate Listing 4.1 at the following columns:

```
1 > ... key|Value|Sequence| = |new |Array|List|<|Data|>|()|;|
2 > ... int| |map|Load|Chunk|Size = |node|Engine|.|get|Group|Properties()|.|
    MAP|_|LOAD|_|CHUNK|_|SIZE|.|get|Integer()|;|
```

Code generation would then be performed at each computed column. As executing the *CodeGPT* model is a time consuming task, the aim of our algorithm is to minimize the amount of requested predictions. A brute-force alternative would truncate the source code lines at every column (i.e. every character). However, such an approach would incur in a lot of computational effort as the number of columns to perform code completion on would increase considerably. As mentioned, the purpose of the truncation algorithm is to save time and effort on the code generation step by reducing the number of code sequences provided to the model, albeit with the drawback that some columns will be missed. In the provided example, the computed columns do not include the ideal one, as Listing 4.1 highlights. However, we shall see ahead that these occurrences are not necessarily a problem and that this program can still be fixed.

4.5 Code Generation

We use the CodeGPT-adapted model to perform open-ended code generation. CodeGPT-adapted inherits the same model architecture of GPT-2, which is a natural language model conceived to perform multiple tasks such as text translation, question answering and text summarization. These tasks imply text generation, which makes GPT-2 a generative model. CodeGPT-adapted is based on GPT-2 as a starting point and is trained on code samples, making it a language model pre-trained for programming language (PL). Two separate versions of CodeGPT-adapted are provided for Python and Java, with the latter being the focus of this work. These models are made available through HuggingFace's Transformers library which provides a Python API.

One of the motivations of this work is to assess how program repair can be seen as a code generation task, more specifically code completion. As such, to perform code completion on buggy programs, we first need to provide a sequence of input tokens to the model. This will be the context to consider to continuously generate new sequences of tokens. After establishing the column to perform code completion on (as per the previous section), we retrieve the previous 1000 tokens and feed them to the model in order to generate the sequence to follow[2]. However, we do not want to limit ourselves to a single prediction and wish to explore several completion possibilities. *Greedy search* generates a sequence of tokens by following the path with the highest probability and *beam search* allows us to explore different hypothesis each time by keeping track of multiple high probability paths. Although *beam search* avoids restricting ourselves to only one completion, it is still based on the tokens with highest probability, making the different generations similar to each other (Holtzman et al., 2020). To circumvent this, we use an indeterministic scenario to produce several completion possibilities. Instead of deciding the next token based on the highest probability, we ask the model to make this selection based on a conditional probability distribution through *sampling*. This way, we introduce randomness in the generation task and have more diversification.

For each column, we consider a sequence of 20 newly generated tokens and repeat this step 10 times in order to produce different code completions for the same input — Algorithm 4.

[2]CodeGPT's context size limit is 1024 tokens, so we need to leave some space for the output tokens. Still, the number of input and output tokens is easily parameterizable in our tool and different values can be explored.

Data: A truncated program TP, a code generation model $CodeGPT$
Result: The list of code completions $Completions$
$Completions \leftarrow []$;
tokens $\leftarrow$ lastTokens(TP, 1000);
foreach $i \in 1..10$ **do**
    completion $\leftarrow$ complete($CodeGPT$, tokens, length=20, sampling=true);
    add($Completions$, completion);
end

Algorithm 4: Algorithm for generating code sequences

4.6 Bounding Code Generation

Code completion is done by using *CodeGPT* to perform code generation. As this process is open-ended, we need to focus on portions of the generated sequences before inserting them in the buggy code. To do this, we defined criteria that bound code sequences at various places, generating different sub-sequences of interest. The boundaries are specified by combinations of characters that are relevant regarding code syntax. Sometimes, the context provided to the code generation model may produce slightly off but almost correct results. As such, it is crucial that we discard elements in the beginning and in the end of token sequences to remove such noise from the nearly accurate predictions.

Skip. In order to reject incorrect tokens from the start of the generated code sequences, we *skip* those characters. The code in Listing 4.2 denotes an example in which the sub-token `EXT` should be removed in order to fix the bug.

```
1  150c150
2  < GL.glGenTextures|EXT(n, textures, Memory.getPosition(textures));
3  ---
4  > GL.glGenTextures(n, textures, Memory.getPosition(textures));
```

Listing 4.2: Change method call — sub-token `EXT` removal

Asking *CodeGPT* for 10 different code completions on the cursor position outputs the following predictions.

```
1  EXT(n, textures, Memory.getPosition(textures));}void glUniform3
2  EXT(n, textures);} public void glTexParameterf (int index, float fval
3  EXT(n, textures, Memory.getPosition(textures));} public void glSten
4  EXT(n, textures, memory.getPosition(textures));} public void glFramebuffer
5  EXT(n, textures, Memory.getPosition(textures));} @Override public void flush
6  EXT(n, textures, Memory.getPosition(textures));} public void glVertex
7  EXT(n, textures, Memory.getPosition(new Integer(n)));}public void
8  EXT(n, textures, Memory.getPosition(textures));} public int nGLObject
9  EXT(n, textures, Memory.getPosition(textures));}void glGetA0
10 EXT(n, textures);}} public void glVertexBegin (int x, int y
```

Listing 4.3: Generated completions for Listing 4.2

Every alternative begins with the undesired `EXT` word. However, the subsequent generated tokens of some completions are able to build the expected code until the character ending the statement: `';'` (semi-colon). The character `'('` (left parenthesis) has special significance in the language — in this case, establishing the beginning of the parameters. As such, we can use this knowledge and *skip* the starting tokens in the predictions until we find the left parenthesis.

Stop. As generated sequences can be potentially unlimited, there is still the problem of determining at what point we *stop* considering the output tokens. Similarly to the previous situation, code generation is performed without considering any syntactic aspects. As such, we again resort to specific characters in order to decide the locations after which we stop incorporating tokens for patch production.

The completions in Listing 4.3 have multiple characters that can be considered for the *stopping* criteria and that will lead to the creation of successful fixes. Taking the first completion line from Listing 4.3, considering the left parenthesis as a *skip* criteria and comma, space, left parenthesis and semi-colon as *stop* criteria would trim the sequence and produce the following possibilities:

```
1 EXT(n, textures, Memory.getPosition(textures));}void glUniform3
2 EXT(n,␣textures, Memory.getPosition(textures));}void glUniform3
3 EXT(n, textures, Memory.getPosition(textures));}void glUniform3
4 EXT(n, textures, Memory.getPosition(textures));}void glUniform3
```

Listing 4.4: Trimmed sequences from the previous completions

Although it may seem the first three alternatives stop earlier than intended, there is still a step to perform in order to fit the trimmed sequences in the buggy code. This last synchronization step will ensure these slices are applied correctly to the buggy code.

4.7 Character Synchronization

The last step to produce a candidate patch consists of fitting the trimmed completions in the buggy code. As explained, completions were altered regarding different criteria to produce adequate alternatives. The final step to correctly incorporate these pieces of code is to synchronize the sequences with the line of code from the original program. Similary to the previous section, this synchronization process is also achieved by coordinating the occurrence of relevant characters of the language's syntax.

Considering the example in the previous section, the first trimmed completion from Listing 4.4 (second line in Listing 4.5) can be incorporated in the buggy program (first line in Listing 4.5) by synchronizing both code sequences on the first occurrence of the comma character, thus producing the desired fix (third line in Listing 4.5).

```
1 bug:   GL.glGenTextures|EXT(n, textures, Memory.getPosition(textures));
2 completion:            |EXT(n, textures, Memory.getPosition(textures));}...
3 patch: GL.glGenTextures(n, textures, Memory.getPosition(textures));
```

Listing 4.5: Produced fix for Listing 4.2

As we can see, the completion is successfully integrated in the original program and we are able to maintain the rest of the code accordingly.

4.8 Experiments

To validate our approach we used the *ManySStuBs4J* dataset (Karampatsis and Sutton, 2020) to conduct a large scale experiment [3]. The dataset version mined from 100 open source Java projects, containing 11624 bugs, was filtered and bugs that did not fit the following criteria were removed:

- line numbers reported in the dataset match the ones obtained through our automated analysis;
- the bug can be repaired only through line changes, i.e. line additions and deletions are not necessary;
- fixes are not produced by changing string literals;
- our truncation algorithm computes at least one column number.

After this filtering step, 6415 bugs remained. For all these bugs, we applied a similar procedure to what was described in Section 5.3. However, there is a difference in the way column numbers were computed. Although our algorithm computes less column numbers than a brute-force approach, applying it to a dataset with such an amount of bugs would make the experiments impractical by taking considerable time to execute. Instead, for each line, we compared the buggy and the fixed version and used the first differing character (column) as the place to truncate the program. Considering Listing 4.2, the cursor position denotes the column used in that case, as that is where characters start differing.

[3] https://gitlab.com/FranciscoRibeiro/manysstubs4j-experiments

Out of the 6415 bugs, our technique was able to fix 1739 programs, representing a 27% effectiveness rate. Even though *random sampling* introduces indeterminism during code completions done by *CodeGPT*, the implementation of our work allows for reproducibility as we fix the seeds for random number generation. This implies that if one of the column numbers computed by the truncation algorithm matches the first differing character, our approach will be able to fix it as the sequence of generated tokens will be the same and the rest of the pipeline is deterministic. The truncation algorithm is able to successfully compute the column numbers used in these experiments for 5674 bugs, which means the algorithm can infer the closest place to the bug 88% of times. Although this step fails to calculate the nearest column number for some cases, it does not mean that our technique is not able to repair them. In fact, performing code completion at different column numbers may still produce a fix. From the 1739 fixed programs, the truncation algorithm did not compute the nearest column for 97 of them. Nonetheless, our technique was still able to fix these programs.

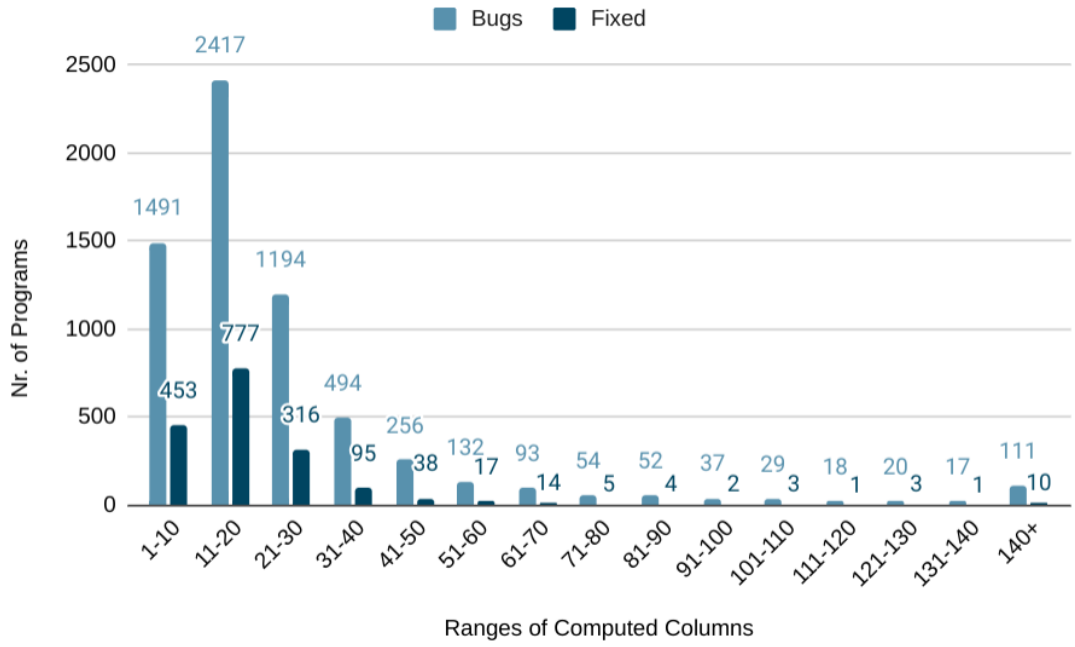


Figure 4.2: Bugs per range of computed column numbers

Even though code completion was not performed on every plausible column, we still applied the algorithm to compute such column numbers to all the bugs in the study. Figure 4.2 shows the number of bugs per ranges of computed columns — size 10 buckets. The algorithm computes between 10 and 20 column numbers for 2417 bugs, representing 38% of the analyzed programs, and we are able to fix 777 of them, which results in a 32% repair rate.

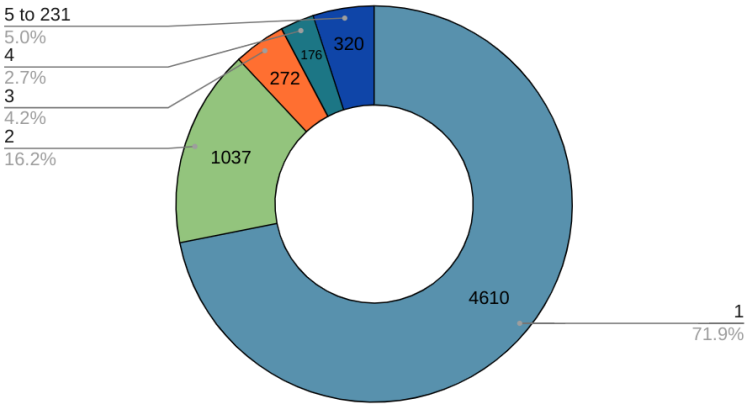


Figure 4.3: Bugs per lines to modify

Figure 4.3 shows the amount of bugs per number of lines. The vast majority of the studied programs — 4610 — are single-line bugs, which consists of 72% of the total amount. Multi-line bugs range from 2 to 231 lines with 1037 and 2 programs respectively. As single-line bugs are the most predominant, it is relevant to focus on this sizable segment of programs to understand how values are distributed.

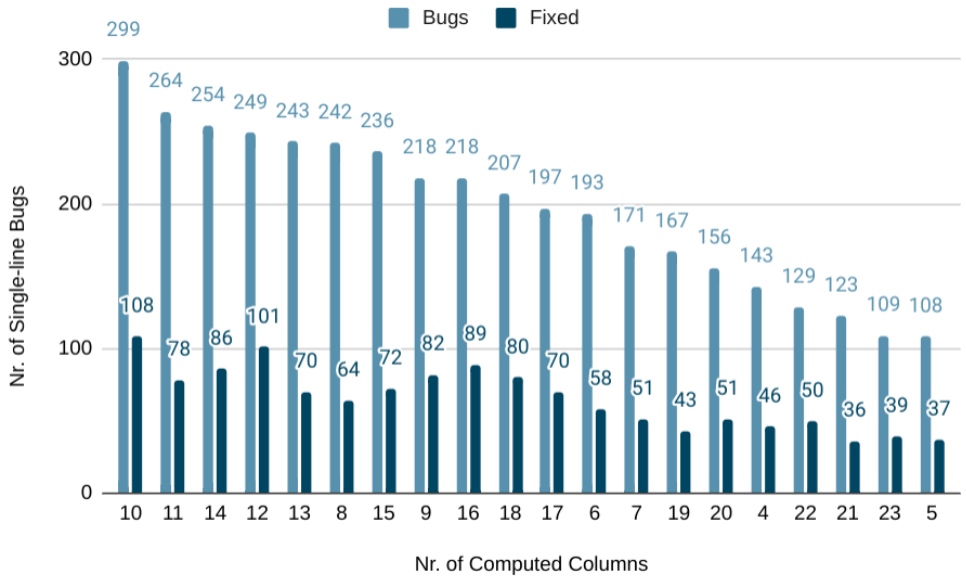


Figure 4.4: Single line bugs per computed column numbers — top 20

Figure 4.4 shows the top 20 computed column numbers for single-line bugs. This data does not present a discrepancy when compared to the distribution illustrated in Figure 4.2 for all the bugs, as the most frequent number of columns are contained within the top-3 buckets. Furthermore, our approach is able to fix 1502 of these programs, resulting in a repair rate of 33%

for this set of bugs. The fact that this segment represents a significant portion of the dataset is also reflected in the number of repaired programs, with 86% of all 1739 fixed programs being part of this group of bugs.

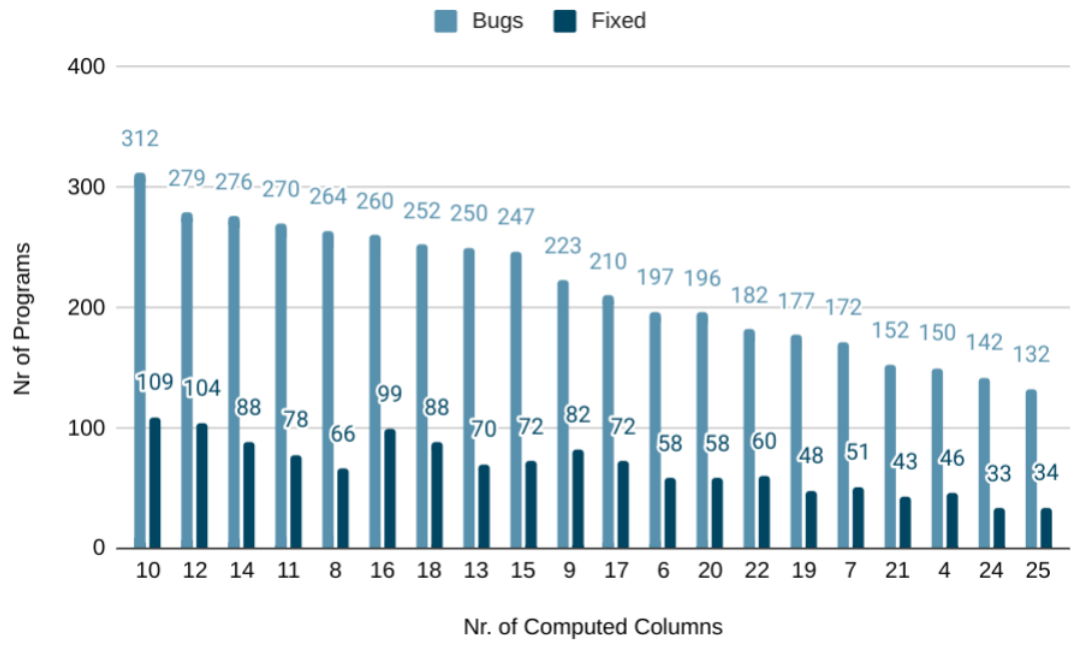


Figure 4.5: Bugs per computed column numbers — top 20

The largest cluster of bugs in the dataset — single-line — has its top-20 most frequent numbers of computed columns in line with the global top-20 — Figure 4.5 — with only the last two places, 23 and 5, missing from it.

4.9 Case Studies

In this Section we present specific examples of programs to explore relevant scenarios that showcase the advantages of our approach.

Case Study 1 Listing 4.6 shows a bug and its corresponding fix.

```
1 404c404
2 < } else if (itemActionLayout >= 0) {
3 ---
4 > } else if (itemActionLayout > 0) {
```

Listing 4.6: Case study 1 — bug and fix

In the buggy line, we can see the column for which code completion should be performed. However, the algorithm defined in this work does not compute that column number because splitting a binary operator does not fit the defined criteria, as Listing 4.7 shows.

```
1  } else if (|item|Action|Layout| >= |0|) {
```

Listing 4.7: Case study 1 — computed columns

Nonetheless, we are still able to produce a fix in this situation by making use of a completion after `itemActionLayout`.

```
1  bug:   } else if (itemActionLayout| >= 0) {
2  completion:                       |!= null) {if (itemShowAsAction > 0) {...
3  patch: } else if (itemActionLayout> 0) {
```

Listing 4.8: Case study 1 — produced fix

Even though *CodeGPT* does not complete the condition with the intended binary operator (>) and operand (0) straightaway, as a result of outlining the generated sequence (Section 4.6), we are able to *skip* (highlighted in grey) undesired tokens and make use of a subsequent comparison from the initial completion. By then *stopping* at the closing parenthesis and discarding tokens coming afterwards, we can synchronize (Section 4.7) the extracted portion of the completion (highlighted in green) with the buggy line and produce the patch shown in Listing 4.8.

Case Study 2 Listing 4.9 shows a scenario for which our technique expanded an existing condition.

```
1  78c78
2  <    return mModelClasses|.size() > 0;
3  ---
4  >    return mModelClasses != null && mModelClasses.size() > 0;
```

Listing 4.9: Case study 2 — bug and fix

As we can see from Listing 4.10, the truncation algorithm computes the closest column to the bug. Therefore, our approach is able to successfully produce a repair for this program.

```
1  |return |m|Model|Classes|.|size|()| > |0|;|
```

Listing 4.10: Case study 2 — computed columns

As results are replicable, we can safely infer *CodeGPT* would generate the same code sequences for that column without incurring in the overhead of applying our approach to every computed column.

```
1 bug:     return mModelClasses|.size() > 0;
2 completion:                  |!=null && mModelClasses.size()>0;}...
3 patch:   return mModelClasses!=null && mModelClasses.size()>0;
```

Listing 4.11: Case study 2 — produced fix

For this example, code completion was able to generate useful tokens right from the beginning. As a consequence of using variable `mModelClasses` to produce the first comparison (`mModelClasses!=null`), the original expression, which constitutes the second part of the condition, is lost. Nonetheless, the continuation of the token sequence generates the desired code (`mModelClasses.size()>0`), eliminating the problem. After discarding tokens to the right of the semi-colon (grey) and using the same character to synchronize the resulting sequence with the buggy code, we produce a patch that fixes this program as shown in Listing 4.11.

Case Study 3 The example in Listing 4.12 illustrates a multi-line bug that we are able to fix by performing the procedure on three separate lines.

```
1  176c176
2  < if(request.get|TaskDefinitionKey() != null) {
3  ---
4  > if(request.getDueDate() != null) {
5  179c179
6  < if(request.get|TaskDefinitionKey() != null) {
7  ---
8  > if(request.getDueBefore() != null) {
9  182c182
10 < if(request.get|TaskDefinitionKey() != null) {
11 ---
12 > if(request.getDueAfter() != null) {
```

Listing 4.12: Case study 3 — bug and fix

For each buggy line, the column closest to the bug is computed — Listing 4.13.

```
1 if(|request|.|get|Task|Definition|Key|()| != |null|) {
```

Listing 4.13: Case study 3 — computed columns

Similarly to the previous case study, as the experiments are reproducible, our pipeline will always produce the same results for this column, which assures us this bug is fixable.

```
1  bug:    if(request.get|TaskDefinitionKey() != null) {
2  completion:           |DueDate()!= null) {taskQuery.dueDate(...
3  patch:  if(request.getDueDate()!= null) {
4
5  bug:    if(request.get|TaskDefinitionKey() != null) {
6  completion:           |DueBefore()!= null) {taskQuery.dueBefore(...
7  patch:  if(request.getDueBefore()!= null) {
8
9  bug:    if(request.get|TaskDefinitionKey() != null) {
10 completion:           |DueAfter()!= null) {taskQuery.dueAfter(...
11 patch:  if(request.getDueAfter()!= null) {
```

Listing 4.14: Case study 3 — produced fix

All three lines are fixed in a similar way. However, the method names that need to be generated are different from each other. Code completion is able to produce the necessary tokens from the start and we only need to *stop* considering the sequence after the first opening parenthesis. Both the buggy line and the code sequence are synchronized also using the opening parenthesis, thus creating the patches seen in Listing 4.14. On the other hand, some of the discarded tokens (`!= null){`) could also be utilized for patch production as they would correctly complete the buggy code. As such, this program could be fixed by using different characters for bounding the generated code sequence, like the closing parenthesis or the opening curly bracket.

Case Study 4 Listing 4.15 shows a bug that is fixed by replacing a method call with a constant.

```
1 272c272
2 < buf.get(bulk, |buf.position(), len);
3 ---
4 > buf.get(bulk, 0, len);
```

Listing 4.15: Case study 4 — bug and fix

Again, the truncation algorithm computes the column nearest to the bug — Listing 4.16.

```
1 |buf|.|get|(|bulk|, |buf|.|position|()|, |len|)|;|
```

Listing 4.16: Case study 4 — computed columns

As a result of reproducibility, the circumstances certify this bug is fixable under our approach.

```
1 bug:    buf.get(bulk, |buf.position(), len);
2 completion:           |0, len); os.write(bulk); } dos.write(buf...
3 patch: buf.get(bulk, 0, len);
```

Listing 4.17: Case study 4 — produced fix

Listing 4.17 shows the produced fix. In this case, the line is truncated at the beginning of the expression that needs to be replaced. However, the necessary expression (`0`) is much different from the original one (`buf.position()`). Nevertheless, the code completion step is able to infer the next tokens correctly from the provided context and the relevant part is extracted accordingly. There is no need to *skip* any unnecessary tokens. Additionally, we can simply *stop* considering the generated token sequence after the comma character and also use it to synchronize with the original code. As in the previous case study, this bug may be fixed in different ways. The remainder of the code sequence (`len);`) can be safely inserted in the original program as it corresponds to an already correct part. For this to happen, the closing parenthesis or the semi-colon need to be used for limiting token generation and synchronization.

4.10 Threats to Validity

Our main goal is to explore if program repair can be treated as a code completion task. More precisely, can we use code generated by DL models such as CodeGPT to evolve faulty programs in order to correct their behavior? We believe our work shows the answer to this question to be yes, though we acknowledge potential challenges to our rationale.

Internal Validity: Some programs need larger and more complex changes in order to be repaired. That is, not all pairs of bugs and corresponding fixes are equivalent in kind. Essentially, this means that different programs need to meet different demands to be classified as correct. However, we consider that the results show our approach was successfully applied to programs of different kinds going from needing small adjustments to multiple intricate changes.

External Validity: The reported results are obtained by analyzing programs from a dataset aiming to provide a collection of single statement bugs. As such, these simple bugs may not be representative of the real-world complexity of software and its needed changes. However, the dataset used in our work was created by extracting actual occurrences from real-world open-source projects, showing that such instances typically arise. In addition, some multi-line changes may be seen as aggregates of multiple single-line modifications. Aside from that, our work targets *Java* programs and, thus, does not encompass a lot of other languages. Nonetheless, many of the language's features and constructs are common to other languages and *Java* is one of the most used by developers.

Construct Validity: Some typical NLP practices were put into place in our work. For text generation, maximizing the probability of the decoded segments leads to poor quality outputs, contrasting with the training objective used to build such models. Higher quality text can be obtained by employing a decoding strategy that uses sampling (Holtzman et al., 2020). Even though these sampling techniques have their roots in NLP, we are convinced we successfully applied them to a PL setting as we were able to fix more programs by not only producing multiple alternatives but also making them more reliable.

4.11 Related Work

Barr et al. (2015) introduce a software transplantation technique transferring behavior from a *donor* to a *host* program. The focused part in the donor is termed the *organ*. The aim is to recreate its feature in an unrelated target program. This approach finds applications in software development, illustrated by transferring a video encoding implementation from the *x264* utility to the *VLC* media player.

Similarly, Shariffdeen et al. (2021) use an equivalent reasoning but focus on transfering patches from a *donor* to a *host*. The authors highlight the technique's usefulness for scenarios in which differing implementations may benefit from patch adaptation.

Transplantation considers a *host* benefits from having a *donor's* feature transferred to it. Likewise, the training process in code generation models like CodeGPT enhances output quality, showing the utility of learning from other programs. Thus, we consider other works' (Barr et al., 2015; Shariffdeen et al., 2021) analogous methods as a validation of our solution.

APR techniques based on neural networks are a clear advance in software reliability. However, some of these (Li et al., 2020; Lutellier et al., 2020; Chen et al., 2021; Ding et al., 2021) focus on partial code snippets and do not acknowledge the entire source code, therefore missing the entire perspective and making learning the code syntax restrictive. Jiang et al. (2021) point these limitations out and build a pre-trained model from a large code repository before perfoming any APR task. By using CodeGPT, our work shares the same logic as we leverage the capabilities of a pre-trained model that first understands the language it is trained on without influence from a specific task beforehand.

Ribeiro et al. (2021) perform fault localization by identifying the semantics behind faults. This is done by translating the AST difference between two program versions — before and after the bug — into mutation operators. The authors are able to infer mutations 78% of times. They demonstrate how real-world programs can be automatically repaired by applying mutation operators that revert the faulty modifications at the inferred places while leaving new but unrelated code unchanged. Likewise, we compute column numbers and consider them the most appropriate spots to generate new code and integrate it.

Pre-trained models like CodeBERT have been fine-tuned on the *ManySStuBs4J* dataset in

order to automatically repair programs and shown to be able to produce patches of variable length and complexity while reporting accuracies between 19% and 72% (Mashhadi and Hemmati, 2021).

4.12 Summary

This work presents an automated repair technique that, given a buggy file and line number, produces candidate patch lines in an attempt to fix the program. We devised a truncation algorithm that computes column numbers for which we use CodeGPT to perform code completion on. After that, we explained our implementation to limit the generated code sequences and how we fit the resulting string based on the language's syntax. Our approach was validated by analyzing the *ManySStuBs4J* dataset. The results show that 1739 programs were fixed out of 6415, which reflects a 27% repair rate and corroborates our work's soundness.

Replication Package

The necessary resources to replicate this study are publicly available:

- **Truncation tool:** `github.com/FranciscoRibeiro/code-truncater`
- **Large scale experiments:** `gitlab.com/FranciscoRibeiro/manysstubs4j-experiments`

LLMs for Type Error Debugging

5

"GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair"
Francisco Ribeiro, José Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva
In: Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering, SLE 2023
Cascais, Portugal, October 23–24, 2023
DOI: **10.1145/3623476.3623522**

Abstract *Type systems are responsible for assigning types to terms in programs. That way, they enforce the actions that can be taken and can, consequently, detect type errors during compilation. However, while they are able to flag the existence of an error, they often fail to pinpoint its cause or provide a helpful error message. Thus, without adequate support, debugging this kind of errors can take a considerable amount of effort. Recently, neural network models have been developed that are able to understand programming languages and perform several downstream tasks. We argue that type error debugging can be enhanced by taking advantage of this deeper understanding of the language's structure. In this chapter, we present a technique that leverages GPT-3's capabilities to automatically fix type errors in OCaml programs. We perform multiple source code analysis tasks to produce useful prompts that are then provided to GPT-3 to generate potential patches. Our publicly available tool,* Mentat*, supports multiple modes and was validated on an existing public dataset with thousands of OCaml programs. We automatically validate successful repairs by using Quickcheck to verify which generated patches produce the same output as the user-intended fixed version, achieving a* 39% *repair rate. In a comparative study,* Mentat *outperformed two other techniques in automatically fixing ill-typed OCaml programs.*

5.1 Introduction

Programming languages usually have an associated type system responsible for determining whether some operation can be applied to some program term. This system ensures a program's correctness in terms of type safety. That is, if a program does not *typecheck*, it signals a logical error related to the inherent type constraints. However, even after knowing there is some type inconsistency, we still need to understand where and why that error occurred. In other words, a type system may be unable to provide the location of the error and the explanation as to why the error arose.

Undeniably, programmers are not completely left in the dark in this regard. Several programming languages provide *type inference* systems, which compute the expected type of expressions in the code. Despite considerable effort (Lee and Yi, 1998; Chitil, 2001; Stuckey et al., 2003; Tsushima and Asai, 2013, 2014; Heeren et al., 2003) to provide helpful information for type error detection, compilers often fail to pinpoint the true cause of an error. Consider the following ill-typed *OCaml* program:

```
1 let rec add_list lst = match lst with
2     | [] -> []
3     | fst :: rest -> fst + (add_list rest)
```

Listing 5.1: Ill-typed function: patterns differ on returned types

Program 5.1 consists of a recursive function `add_list` that takes a list of integers and should calculate the sum of all numbers. `ocamlc` would yield the following message[1]:

```
1 3 |              | fst :: rest -> fst + (add_list rest)
2                                        ^^^^^^^^^^^^^^
3 Error: This expression has type 'a list but an expression was
     expected of type int
```

The type system successfully detects a type error in the program and the compiler provides a message reporting the problem. If we replace the use of the *plus* operator (+) in line 3 by the *cons* operator (`::`), the whole program is well-typed. However, the expression highlighted as being problematic by the compiler is not the true origin of the error. As a consequence, the information about the mismatch of the expected type (`int`) and the inferred type (`'a list`) does not provide meaningful advice into how to approach the problem. Another way to fix this program, which corresponds to the programmer's intended fix, is to have it return zero (0) instead of the

[1] In *OCaml*, polymorphic type names are prefixed with a backquote.

empty list ([]) in line 2. Hence, it often happens that the user's intended modification differs from what the compiler points out. That is because reporting type inconsistencies is influenced by the order in which expressions in a program show up. As long as no inconsistencies are detected, the inferred type for an expression is considered to be correct. As a result, the type system will have a *left-to-right* bias and errors tend to show up towards the end of a program (Heeren et al., 2002). Now consider that we swap the two patterns:

```
1 let rec add_list lst = match lst with
2     | fst :: rest -> fst + (add_list rest)
3     | [] -> []
```

This time, we get a different error message:

```
1 3 |                   | [] -> []
2                                 ^^
3 Error: This expression has type 'a list but an
     expression was expected of type int
```

This means that the type error we are dealing with can have multiple causes. Depending on the order of the patterns, the cause that is reported changes.

However, even after recognizing the inherent limitations of type systems in accurately locating and explaining type inconsistencies, we are still left with fixing them. Automated program repair (APR) aims to generate patches for incorrect programs (either syntactically or semantically) with minimal human intervention (Goues et al., 2019). Many approaches have emerged based on the competent programmer hypothesis or, put in other words, programmers "create programs that are close to being correct!" (DeMillo et al., 1978). We argue that automatically finding repairs that eliminate type inconsistencies is one effective way of locating and understanding the root of a type error.

In this chapter, we present an approach that leverages the code understanding and generation capabilities of models based on GPT-3 to automatically fix type errors in *OCaml* programs. We focus on analyzing the source code of ill-typed programs and generating prompts that are then provided to the model. By doing this, we aim to produce programs free from type errors and, thus, can be used to find and understand what was causing them. Our contributions are:

1. a source code analysis and manipulation technique that produces different kinds of prompts intended for GPT-3-based models (Section 5.3);
2. a publicly available tool, named Mentat, implementing this technique (Section 5.4);

3. an initial validation on a small set of programs, followed by a large-scale evaluation on an independent repository, with an analysis of the results obtained (Section 5.5);
4. a comparative study between our tool, Mentat, and two other techniques, namely Rite (Sakkas et al., 2020) and Seminal (Lerner et al., 2007) on a common dataset, alongside the obtained insights.

Even though this work is concerned with type error debugging, it differs from previous approaches, which aim to improve the quality of type error messages (Damas and Milner, 1982; Wand, 1986; Lee and Yi, 1998), provide interactive type debugging (Chitil, 2001; Tsushima and Asai, 2013; Chen and Erwig, 2014a,b), and narrow down the area for type error debugging (Haack and Wells, 2004; Rahli et al., 2017; Stuckey et al., 2003, 2004; Schilling, 2012). Instead, our work focuses on the automatic repair of type errors. We achieve this by analyzing and transforming source code and outputting it in a form that can be understood and processed by GPT-3, a large language model trained by *OpenAI*.

For the initial validation, we find that our tool presents at least one valid solution for each test program, with the *Fill* operation mode obtaining success rates varying from 53% to 60% for simple programs and from 83% to 100% for implementations of the *shortest path* algorithm — also referred to as the *Dijkstra* algorithm from here on out (Dijkstra, 1959; Frana and Misa, 2010). Regarding the large scale evaluation, we analyzed $1{,}318$ buggy programs and were able to fix 516 of them, reaching a 39% repair rate. To automate this process we used two key features of property-based testing (Claessen and Hughes, 2000): firstly, we automatically generate a very large number of random inputs, and secondly we define a property that tests whether the user-fixed program outputs the same result as the automatically repaired one. The program-specific property is also automatically generated, thus having a fully automated large scale validation process without relying on human intervention to inspect the generated patches. Also, we showed the potential for partial fixes by considering the results for programs that do not pass 100% of test cases. While the other operation modes perform worse overall when compared to *Fill*, they are still capable of generating successful results and, in some cases, succeed where *Fill* fails. Moreover, we performed a comparative study of our technique with two type repairing approaches, namely Rite (Sakkas et al., 2020) and Seminal (Lerner et al., 2007). Our first results show that Mentat gives the best program repair results with a 37.5% repair rate versus 33.4% from Rite and 7.8% from Seminal.

5.2 Background

A *type system* is a set of rules governing how data is represented and used in a program (Pierce, 2002). It is lightweight and does not require special knowledge from the user. *Type inference* is a static algorithm to find the type of each part of a program without the programmer's annotation. The advantages of type systems include not only type-safety of programs but also efficient computation by enabling the generation of optimized code. For this reason, *Ruby*, a dynamically typed language, has recently introduced type inference, and *Python* has also introduced type-related features.

Originally, research in natural language processing (NLP) focused on ways of processing, analyzing, and manipulating natural language through statistical and rule-based modeling. More recently, the use of artificial intelligence has allowed the development of techniques that make NLP one of the most prominent fields in computer science. Simply put, NLP is responsible for encoding text into more appropriate machine level representations and also for processing and transforming these lower level descriptions into other forms of text. More specifically, neural networks have been very impactful in the development of NLP models. Some of the most important ones are BERT (Devlin et al., 2019) and GPT (Radford et al., 2019) which have seen their utility displayed in an overwhelming amount of more specific downstream tasks. Encouraged by the achievements conducted in this area, the software engineering community has successfully applied some of NLP's fundamentals to build tools that improve software development workflow. Code completion is one of the most popular features and many code editors implement it one way or another. With the intent of going beyond basic level completions like more pertinent suggestions for API calls for a given context, the research community has also directed its efforts into developing versions of the BERT and GPT models that are specifically tailored towards programming languages. New models, such as CodeBERT (Feng et al., 2020) and CodeGPT (Lu et al., 2021), were created based on the original architectures. GPT-based code models are able to generate long and relatively complex code sequences by analyzing and inferring the context of the source code provided as input. One of the most recent iterations of such models is GPT-3 (Brown et al., 2020), which presents a high degree of success when employed in different scenarios such as cloze and completion tasks.

5.3 Technique

With our contribution, we intend to, for a given faulty *OCaml* program, extract as much information from it as possible, and then format it in a way that GPT-3 can understand and process it. For this, we first check for type inconsistencies. If they are present, we employ three different tasks: *type error location*, *inlining*, and *type unification*, with each being described in detail in the following sections. Figure 5.1 illustrates how the tasks generally interconnect and highlights them with the corresponding label. Nodes with dashed borders represent steps in which we make use of existing components and are not directly part of our contribution. Grey nodes and white nodes represent elements and actions, respectively. Depending on the way we wish to interact with GPT-3, the tasks may be combined in slightly different ways.

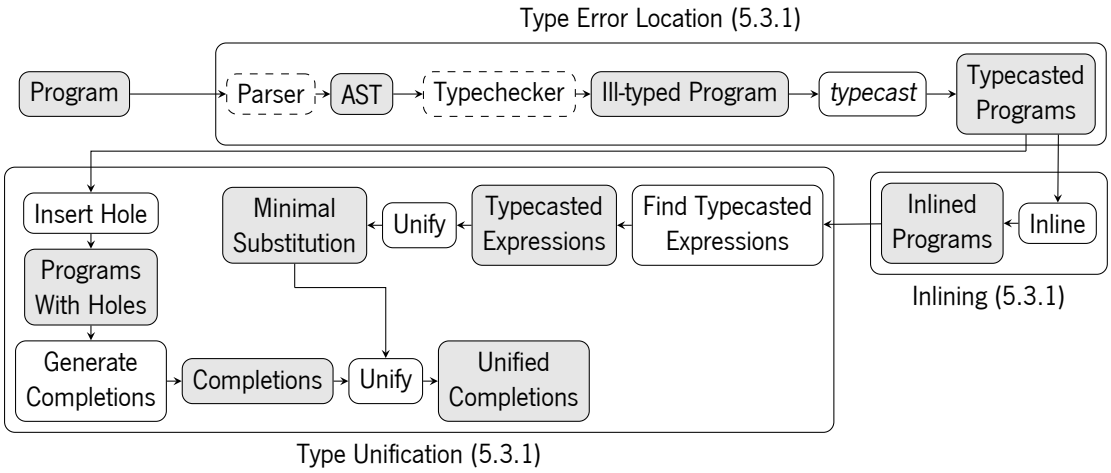


Figure 5.1: Interconnection of source code manipulation tasks

5.3.1 Source Code Analysis

Type Error Location

The compilers of strongly typed programming languages tend to check for source code errors in two separate steps when building an executable file: parsing and type checking. The parser checks whether the input is syntactically correct and if so it produces an *AST (Abstract Syntax Tree)*. The type checker traverses such a tree to check whether the underlying program obeys the type rules. If it does parse but it does not typecheck, then there is a type error, which is the

focus of this work. If it does both parse and typecheck then, for our purposes, the program is considered correct.

Some programming languages offer us type conversion and manipulation tools; we focus on type conversion tools from strongly typed languages. Let us consider a function from now on referred to as *typecast*, which forcefully converts a value of any type into a value of any type. Of course, such function will not be able to actually do these conversions during a program's runtime, but it will be able to trick the compiler into interpreting an expression of one type as if it had a different type. As such, the *typecast* function will only be used when typechecking a program. In *OCaml*, this operation can be performed by using `Obj.magic`[2], which we will use in our tool. Any program referred to as *typecasted* from here onwards is a program in which part of it was transformed with the *typecast* function.

Recall the introductory example in Program 5.1. Because any type can be converted into any type with this function, we can, for example, apply *typecast* to the empty list (`[]`) to transform it into a different type *'b*, which the typechecker will deduce to be *int*. We could also apply *typecast* to the *plus* operator (+) thus transforming it into a function of type *'b* which the typechecker will deduce to be *'a → 'a list → 'a list*. Finally, we can also *typecast* the expression on the right-hand side and have the typechecker infer the type *'a list*. After parsing the *OCaml* program, we create multiple program variants, each with the *typecast* function applied to a single expression, and then type check each application. Every variation of the original program that typechecks correctly is stored; if changing the type of one value / expression with *typecast* fixes the program type-wise, then we consider that the replaced value can be the error that needs to be fixed. Finally, we replace the usage of this function with a mask, signaling a hole in the program that needs to be filled. The following tasks will focus on analysing and transforming the program variations (which we call the *typecasted* programs) produced in this task.

In this work, the focus is on type errors with a single location. Nonetheless, our approach is still flexible to some instances of errors with multiple locations if all of them are contained within a single function call expression.

[2] `Obj.magic` has the type *'a → 'b*

Inlining

We make use of *inlining* not for the usual purposes of compiler optimisations, but to be able to make information available from some parts of the program in other places. That is, the *inlining* step we describe here is done on the actual source code to improve its analysis further ahead, and not to produce more efficient machine code. This step is particularly useful as it allows us to extract better results from later type unification and type inference tasks. Consider the following *typecasted* program:

```
1 let f x = (Obj.magic (&&)) x (x + x)
```

Listing 5.2: Obj.magic hides the error from the type system

If we ask for the type of the `Obj.magic (&&)` expression, the type system will infer it to be *int* → *int* → *'a*. However, let us now extend the program with a test case:

```
1 let f x = (Obj.magic (&&)) x (x + x)
2 let t = (f 1) = 3
```

The second line specifies the usage of function `f`. By inlining function `f`, we associate it to a context in which the type system can take advantage of the extra information provided by the *int* parameter. As a result, the inferred type of the *typecasted* expression would be *int* → *int* → *int*.

To accomplish this *inlining* step, an environment is maintained throughout the underlying *AST* traversal. When a new definition is found, its identifier is stored and associated to the corresponding expression. As such, when the usage of an element stored in the environment is detected, the usage of the identifier is replaced by the expression's body, effectively *inlining* that piece of code. Special care needs to be taken for two scenarios: recursion and function arguments. For the first one, we need to avoid repeatedly *inlining* the same element as that could potentially lead to a non-terminating procedure. Nonetheless, there is still interest in performing this step once for recursive definitions. Thus, we allow *inlining* to happen exactly once in such cases. For the second scenario, most programming languages allow re-definition of variables in different scope levels and *OCaml* is no exception. It is possible to have a variable `x` already defined, and still define a new `x` in an inner scope. When inlining, in this case, we take care to inline the correct definition for the correct `x` variable. The inlined source code is only stored in memory and the original program is not modified, with GPT-3 never seeing the inlined version.

Type Unification

We make use of type unification to filter elements from a list of completion suggestions. Almost every code editor provides the ability to have completion suggestions on request from the user by specifying a place in the source code. Prior to requesting a list of completion suggestions, we replace *typecasted* expressions (obtained as described in Section 5.3.1) with typed holes. Then, we make use of an *OCaml language server (LSP)*[3] to automatically locate the introduced typed hole and to obtain a list of suggestions containing code elements that may fit. Let us consider Program 5.2 and its equivalent version with a typed hole represented by the underscore:

```
1 let f = fun x -> _ x (x + x)
```

Having introduced the typed hole, we can request a list of suggestions for the typed hole's location from the *OCaml LSP* and obtain 318 candidates. This list is not curated according to the type context and, as such, the suggested completions may present a type mismatch. In order to filter the list accordingly, type unification is performed between each element and the expression that was flagged as problematic according to the application of *typecast*. If unification succeeds, the suggestion will take part in the resulting list which, in this situation, will consist of 23 candidates.

5.3.2 Strategies

We make use of the available GPT-3 operation modes to implement three of the four repair strategies supported by our tool. Depending on which strategy we intend to use, we have to prepare and format the data accordingly.

Fill

GPT-3 provides an operation mode named *Insert*, in which, given some input text from the user which contains a hole denoted by the `[insert]` tag, a generation is produced by the model and placed where the tag was located. Thus, this operation mode is perfect for our use case, by filling programs in which a part is missing. There are several models available for this operation mode, notably `text-davinci-003` and `code-davinci-002`, the former being a general

[3] https://github.com/ocaml/ocaml-lsp

model and the latter being optimized for handling code. Next, we show an example of an input prompt for this operation mode:

```
let rec add_list lst = match lst with
    | [] -> [insert]
    | fst :: rest -> fst + (add_list rest)
```

For this prompt, using the web interface and with the default model parameters, both `text-davinci-003` and `code-davinci-002` models will output the following:

```
let rec add_list lst = match lst with
    | [] -> 0
    | fst :: rest -> fst + (add_list rest)
```

For this program, we obtain the correct patch. Of course, to do so we need to first locate the error and replace it with the `[insert]` tag. We do this through the technique described in Subsection 5.3.1, by replacing the code that did not typecheck without the usage of the *typecast* function with said tag. This is the strategy in which we provide the least information to the GPT-3 model.

Choose

GPT-3 provides an operation mode named *Complete*, in which, after being fed input text from the user, it will attempt to generate more text based on it, that is, complete it. It can be used for non-code tasks such as writing stories or classifying tweets, as well as code tasks like translating plain text to an SQL query. We experimented with several approaches for usage of the *Complete* mode, because, unlike with the other operation modes, there is no intuitive way to use this mode to correct programs. Failed approaches include asking the model to rewrite the entire program replacing the missing hole (similar to the `[insert]` tag mentioned in Subsection 5.3.2) with the correct solution, or asking the model to just give us the code expected in that hole. Variations of this approach, by providing more clues, such as the expected type of the result, or by providing possible solutions to consider, were also unsuccessful. Ultimately, we were able to obtain favourable results by formatting the input as an exercise, similar to what would be found in a student exam. To do this, we present the source code with a missing hole denoted by the `<mask>` identifier, and a list of possible solutions. This list is produced as described in Subsection 5.3.1 and presented as numbered options, and the model is asked to select the most appropriate. We guide the model into selecting one option through *prompt engineering*. Specifically, every

produced prompt is preceded with two example exercises that share this template but have the correct option selected (omitted in listings for brevity). The following program is an example of a prompt formatted for the *Complete* operation mode, as described.

```
Consider the following OCaml program:

let rec add_list lst = match lst with
  | [] -> <mask>
  | fst :: rest -> fst + (add_list rest)

Which of the following options should replace <mask>?
1) ( __LINE__ )
2) ( max_int )
3) ( min_int )

Correct option:
```

Notice that all presented options are incorrect - this is a limitation from using this kind of prompt. Because we are using the *OCaml LSP* to generate suggestions to be then presented here, we are limited in which suggestions can be included. In fact, the *OCaml LSP* will not generate common constant values such as `0`, which is the correct response here. All the listed suggestions are integer constants suggested by the *OCaml LSP*, where `__LINE__` is a compiler macro representing the code line number where it is written, and `max_int` and `min_int` are constants representing the maximum and minimum values possible to represent as integers in *OCaml*. The following is another prompt (re-formatted for brevity) we produce for the same program, but assuming an error in a different place.

```
Consider the following OCaml program:

let rec add_list lst = match lst with
  | [] -> []
  | fst::rest -> <mask> fst (add_list rest)

Which of the following options should replace <mask>?
1) ( fst )                    8)  ( raise_notrace )
2) ( ! )                      9)  ( snd )
3) ( exit )                   10) ( @@ )
4) ( failwith )               11) ( max )
5) ( input_value )            12) ( min )
6) ( invalid_arg )            13) ( List.cons )
7) ( raise )                  14) ( @ )

Correct option:
```

Notice that the list of suggestions grew — some of them, such as `exit` and `raise`, will match a lot of types, due to the polymorphic nature of these suggestions. However, some interesting suggestions are now listed, and in this case, the model suggests option **14** - the `@` operator. This

operator concatenates two lists, and placing it into the hole in the source code will transform this function into a correct implementation of the `List.concat` function for joining a list of lists of values into a single list of values. It is, however, not what we tend to expect out of a function named *add_list*.

Communication with GPT-3 for this operation mode is fairly similar to other modes, but we have to limit the number of generated tokens, as the model tends to try to generate an explanation for its answer. It is also possible to specify a *stop sequence*, which is a sequence of tokens that, when generated by GPT-3, stops the whole generation process.

Instruct

Another way of interacting with GPT-3 is through its *Edit* mode which expects two inputs: a prompt and instructions describing how to edit the prompt. Similarly to the other modes, there is a more general textual model and a code specific variant. However, for this mode, there are specialized versions to handle text editing, namely `text-davinci-edit-001` and `code-davinci-edit-001`. Our approach uses a simplified form of the *Instruct* mode, which is applied when the step in Section 5.3.1 fails to produce a program that typechecks. In that case, the prompt consists of the original program, and the instruction will hold the message "*Fix the bug*". Alternatively, in case the previous step is able to produce a well-typed program[4], our approach performs inlining and type unification on the *typecasted program* in order to compute the minimal substitution holding the expected type with as much information as possible from the whole program. If we consider Program 5.1, the inputs sent to GPT-3 would be:

```
Prompt:
let rec add_list lst = match lst with
    | [] -> _
    | fst::rest -> fst + (add_list rest)
Instruction:
Replace the underscore with something of type int
```

The hole represented by the underscore is the place we wish to see filled in. Although the underscore character can appear in an *OCaml* program, we did not notice any interference in the ability of GPT-3 to apply the transformation in the intended place. The template we use for the edit instructions is "*Replace the underscore with something of type* `<inferred>`". For

[4]Recall that whenever bypassing the type system by using the *typecast* function eliminates the type error, we explore that program variant.

this program, GPT-3 responds with `0`, which is the desired fix. Indeed, it may look like GPT-3 simply understands the program in question is missing the most adequate stop criteria and just answers with a corrected version, perhaps disregarding our instructions. However, the unification and inference step we perform in order to complete the message template with the expected type plays a crucial role. For the same example, fabricating messages referring illogical types such as *string* or *('a → 'b) → 'a list → 'b list* would see GPT-3 answer with the *empty string* and `List.map` respectively.

Because our approach explores every application of *typecast* that typechecks a program, we also produce another alternative:

```
Prompt:
let rec add_list lst = match lst with
  | [] -> []
  | fst::rest -> _ fst (add_list rest)
Instruction:
Replace the underscore with something of type 'a -> 'b list -> '
   b list
```

Even though this alternative prompt will not generate the intended fixed program, it shows that the creation of adequate prompts is essential for GPT-3 to perform well. In this case, GPT-3 will respond with `(fun x y → x::y)`. Surely, integrating that piece of code into the original program produces a correct one from a typechecking perspective, although it does not fulfill the programmer's intention.

Without GPT-3

One interesting outcome from the implementation of the *Choose* strategy described in Section 5.3.2 is that we can make use of the work done to construct the prompt and skip the interaction with GPT-3. Thus, we provide a way to work completely offline. After coming up with an alternative program that typechecks and a list of suggestions (according to sections 5.3.1 and 5.3.1, respectively), we integrate each one into the original program. If no test cases have been provided, the tool simply displays which options fit the expected type. If there are test cases, the tool tests each suggestion and displays the resulting programs according to whether they satisfy the tests or not. Consider the following ill-typed program and the associated test case:

```
let f = fun x -> x && (x + x)
Test case: f 3 = 9
```

According to the test case, the intended fix consists of replacing the logical-and (`&&`) with the plus operator (+). For this case, our tool is able to filter 15 suggestions out of the 318 provided by the *OCaml LSP*, with one of them being the desired one. Each of the 15 suggestions is checked against the test case and the tool outputs the only program that satisfies the criteria:

```
1 let f = fun x -> (+) x (x + x)
```

Note that the type unification step requires the use of functions. In that sense, we convert the usage of operators such as '`&&`' to their equivalent prefix notation functions '`(&&)`', resulting in the generated patches also being written in this form, demonstrated by the use of the function '`(+)`'.

Indeed, the focus of our work is to evaluate GPT-3's performance regarding the automatic repair of type errors, and presenting a method in which the usage of the model is non-existing may seem counter-intuitive. However, we find this to be a validation of our approach, showing that the effort to assemble the prompt can guide the whole process towards the intended result as the correct patch may be found by further checking each plausible option.

5.3.3 Model Bias

We now experiment with providing more information, trying to guide the models into more relevant results. We do this by using the *bias* parameter which lets us guide the model's output by specifying the importance of certain tokens[5] through weights. A token represents a unit of text, like a character or a word. We use a tokenizer tool for this purpose.

We create a database of the most common tokens in the top 10 *OCaml* repositories on *GitHub* programmatically. To achieve this, we utilize GPT-3's tokenizer, which converts text into numerical sequences that the model processes. We analyze the tokens in source code files from these repositories and collect frequency data to construct a database of commonly used tokens in real-world programs.

We create a list of suggestions as per Section 5.3.1, convert them into token sequences, and assign positive weightings to these tokens. Then, we use the *bias* parameter in GPT-3 to guide the model toward these suggestions. The weightings are determined based on a database of token frequencies from real-world programs. We heuristically set minimum and maximum bias

[5]Tokenizer available at `https://beta.openai.com/tokenizer`

values at 1 and 3, respectively, to ensure effective guidance without extreme behavior. Figure 5.2 provides an overview of this process with sample values.

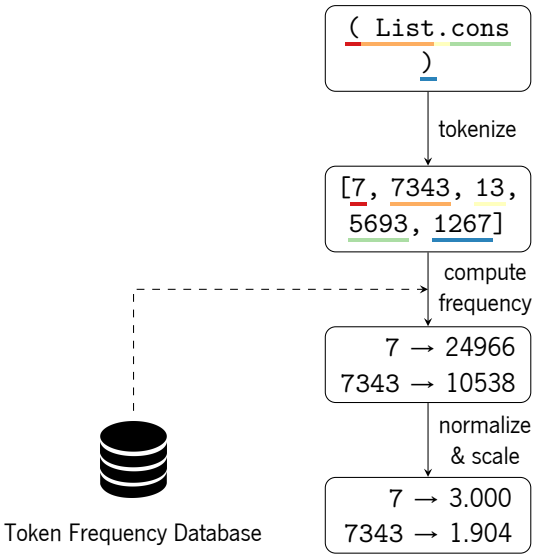


Figure 5.2: Bias computation for one *OCaml LSP* suggestion.

We experiment with *bias* values by comparing the *Choose* strategy with and without *bias*. In the *Choose* strategy with *bias*, we exclude suggestions from the textual prompt since their influence is already provided through *bias* values, as we show in the following example:

```
Consider the following OCaml program:

let rec add_list lst = match lst with
  | [] -> <mask>
  | fst :: rest -> fst + (add_list rest)

What should replace <mask>?

Answer:
```

5.3.4 Test Cases

Mentat allows including test cases when repairing a program. This additional information enhances the system's performance by narrowing the error search space and tightening the type constraints for the function under examination. To illustrate, recall Program 5.1, which contains two potential errors. Now, let's add a test case into this program.

```
1 let rec add_list lst = match lst with
2     | [] -> []
3     | fst :: rest -> fst + (add_list rest)
4
5 Test case: add_list [1;2;3] = 6
```

The newly added test case locks function `add_list` to specifically receive a list of integers and output a single integer. Because of this, the function now only has one possible source of error, which is the empty list (`[]`) in line 2. Previously, it was also considered that the *plus* operator (+) could be a source of error, but with the additional restrictions imposed by the test case, this is no longer possible.

Adding at least one test case to the framework also helps classify GPT-3's generations. After repairing a program, we can use the test case to check for type consistency and verify if it now passes the tests. For instance, in the case of this program, the correct fix would be to replace the empty list (‘[]‘) with the number 0, but substituting it with any other integer would still pass the type-check, although it might produce incorrect results during testing.

5.4 Tool

To validate our approach, we implemented it as a publicly available tool called Mentat [6]. This tool, written in *OCaml*, can analyze *OCaml* programs and is accessible via the command line. Users can specify:

- the file containing the *OCaml* program to analyze;
- the repair strategy by issuing the corresponding flag;
- optionally, one or more test cases that should be satisfied.

Depending on the repair strategy selected by the user, Mentat interacts with GPT-3 by calling the relevant function and setting appropriate parameters. Interaction with *OpenAI's* GPT-3 like models requires an internet connection to use the API. Mentat handles these requests and processes the responses to generate potential fixes for type errors. The resulting programs are saved for further offline analysis, including whether they compile successfully and pass provided test cases if available. Installation and usage instructions are provided in the tool's repository.

[6] `https://gitlab.com/FranciscoRibeiro/mentat`

5.5 Experiments

We benchmark the effectiveness of our tool by running it against several *OCaml* programs containing type errors. For this, we run each strategy 3 times for each program, and record the results. All the examples and necessary resources to replicate the experiments are publicly available[6].

5.5.1 Simple Programs

This set includes 15 ill-typed programs sourced from an introductory *OCaml* class at a Japanese University and the type-error slicer *Skalpel* (Rahli et al., 2017), and previously used in a type-error debugger (Tsushima et al., 2019). These programs are simple, with issues like returning empty lists instead of sums, confusion between `Float` and `Int`, and using values when singleton lists were expected. They range from 29 to 117 tokens and consist of 2 to 8 lines of code. One could argue that simple programs are easier to fix because they are simple, or harder to fix due to the limited contextual information available.

For the text and code models used in the experiments, we use the default parameters (*temperature* of 0.7 for text and 0 for code, and *top_p* value of 1 for both). These settings were found to be the most suitable through extensive testing.

We present the experiment results in Table 5.1. Each test program was processed 3 times to measure successful patch generation, ensuring it passed at least one test case. We employed different repair strategies with models optimized for *text* (T columns) and *code* (C columns). The *C + Bias* column includes additional experiments detailed in Section 5.3.3. The rightmost column represents results without language models, measuring how many suggestions enabled program compilation and passed a test case. For example, program $S2$ was exclusively repaired by the code variant of the *Fill* strategy, with 10 successful repair suggestions that passed the test case. Further refinement may be possible by using different or additional test cases. In each column, we calculate two success rates: *%Repair*, indicating partial success (yellow or green), and *%Test*, indicating total success (treating yellow results as failures).

Each cell of the table is coloured red, yellow, or green. Red cells denote a total failure of patch generation, green cells denote the generation of the correct patch, and yellow cells denote partial success. Examples of patches that are categorized yellow include generating the

Table 5.1: Automatic repair results for 15 simple test programs.

Test Prog.	Fill		Choose			Instruct		No GPT-3
	T	C	T	C	C + Bias	T	C	
S1	3	3	3	3	3	3	3	3
S2	0	3	0	0	0	0	0	10
S3	3	3	0	0	3	3	3	3
S4	3	3	0	0	0	2	3	0
S5	3	3	0	0	3	3	3	2
S6	3	3	3	3	0	3	3	3
S7	3	3	0	3	3	0	0	0
S8	3	0	3	3	3	3	3	1
S9	3	3	3	3	0	0	0	1
S10	3	0	3	3	0	0	0	1
S11	3	3	0	0	3	3	3	0
S12	0	3	3	0	3	2	3	0
S13	3	3	3	3	3	0	0	1
S14	3	3	3	3	3	0	0	2
S15	3	3	3	3	3	3	3	1
%Repair	87%	87%	60%	60%	66%	56%	60%	–
%Test	53%	60%	47%	47%	40%	49%	53%	–

incorrect arithmetic operator (such as generating a *minus* sign when a *plus* sign is expected, or not generating the correct constant value when one is expected) - in some cases, for example when a constant value is expected, it might be completely impossible to reasonably deduce which value the developer expects. Such results can be adjusted by using different test cases which favourably guide the GPT-3 model - for example, if, for a given test case, using a *plus* sign yields the same result as using a *minus* sign, perhaps changing the test case will make the usage of a different operator yield a different result. Nevertheless, we decided to not fine-tune the test cases to maximize result quality, as that is not always realistic.

The results showcased in Table 5.1 point towards the *Fill* strategy being the most efficient for automatic generation of patches. Most notably, all modes have a *%Repair* success rate above 50% and a *%Test* success rate above 40%, and all test programs were successfully repaired by at least one of the repair strategies. This fact points towards the combination of strategies being a robust approach to leverage the strengths of each other. We also denote that most cells contain the values 3 or 0, with rare occurrences of 2, which implies that the model tends towards the same results in different iterations. For this, we have experimented with different values of the parameters we supply to the model, focusing mainly on the *temperature* as it should change its randomness. Nevertheless, the results were not noticeably better, generally leading to lower overall success rate.

We observe that the usage of *bias* with the *Choose* operation mode yields relatively similar results in terms of success rates for these problems. The main difference when using *bias* lies in the fact that some programs that were not repaired with the previous approach are now able to be repaired and vice-versa. For this set of programs, we conclude that the usage of *bias* does not improve the results significantly, but it is capable of generating solutions complementary to the ones generated by the original *Choose* repair strategy.

5.5.2 *Dijkstra* Algorithm

In this set, we have longer and more complex programs for the *Dijkstra* algorithm (*shortest path* algorithm), each with around $2{,}300$ tokens and 170 lines of code. Deliberate errors were added to make the repairs more challenging. We followed the same methodology as in Section 5.5.1 for the results.

Table 5.2: Automatic repair results for 10 *Dijkstra* programs.

Test Prog.	Fill		Choose			Instruct		No GPT-3
	T	C	T	C	C + Bias	T	C	
D1	3	3	0	3	3	0	0	0
D2	1	3	0	0	0	0	0	0
D3	1	3	0	0	2	2	3	0
D4	3	3	3	3	0	1	3	1
D5	2	3	0	3	3	2	3	1
D6	3	3	0	0	0	0	0	0
D7	3	3	3	1	3	3	3	1
D8	3	3	0	0	3	3	3	0
D9	3	3	3	3	0	3	3	0
D10	3	3	0	0	3	0	0	0
%Repair	83%	100%	30%	43%	57%	47%	60%	–
%Test	83%	100%	20%	23%	20%	47%	60%	–

Table 5.2 summarizes the results for this program set. Like in the previous set (5.5.1), *Fill* remains the most effective strategy with an 83% repair rate for the *text* model and 100% for the *code* model. Despite the increased program complexity, *Fill* performed better, with a higher rate of programs passing the provided test cases. Conversely, the other strategies, *Choose*, *Instruct*, and *No GPT-3*, were less effective with this program set. Indeed, depending on the considered repair strategy, the discrepancies across the different sets of programs move in opposite ways. Increased program complexity may have improved *Fill*'s performance by providing more context for the model, while negatively affecting the other strategies, which seem more suited for shorter and simpler repairs.

Compared to the simpler programs in the previous section, the type errors in this set usually need more elaborate repairs. As an example, consider the ill-typed excerpt from a program contained in these experiments and its intended repair:

```
let rec search tree k = match tree with
    Empty -> raise Not_found
  | Node (left, key, value, right) ->
      if k = key then value
      else if k < key then left (* intended: search left k *)
      else search right k
```

Instead of `left`, the intended expression is `search left k`. These repairs need an aggregation of several terms, which is impossible to obtain with suggestions from the *language server*.

Essentially, this severely hinders the *Choose* and *No GPT-3* modes, as they heavily rely on that list of code completions. The *Instruct* mode correctly infers the corresponding hole's type to be *(string * float) list*[7] but is unable to generate the call `search left k` and produces the empty list instead, which, nonetheless, produces a correctly typed program. From these experiments, we take that *Fill* works best for longer programs, as the pure context of the code seems to be enough and better allows GPT-3 to understand and reason about the program at hand. Extra analysis of the source code prior to providing the programs to GPT-3 is more helpful for smaller programs, in which naturally occurring context lacks. This is evidenced by programs $S4$, $S5$, $S8$, $S9$, $S10$ and $S13$, for which *Fill* presented incorrect or only partially correct results, while *Choose*, *Instruct* or *No GPT-3* were able to generate intended outcomes. This did not occur for the *Dijkstra* programs, as *Fill* showed that it could match the effectiveness of the other strategies for each case.

5.5.3 Large Scale Evaluation

We also conducted a large-scale evaluation of our approach. We analyzed a repository of $4{,}500$ *OCaml* programs, which had already been created as part of Rite (Sakkas et al., 2020). We provide detailed analysis of the results obtained from this evaluation, such as the total repair rate, the number of partially fixed programs and the distribution of effectiveness of the three repair strategies. Through this evaluation, we aim to demonstrate our tool's applicability in real-world scenarios and potential to improve the quality and reliability of large-scale software systems.

Pre-Processing the Data

To ensure a comprehensive and accurate evaluation of our tool, we applied a filtering process to the original dataset obtained from the Rite project. Specifically, we filtered out bugs that required modifications in multiple and disjointed places in the code, as the current version of our tool considers single expression bugs, only. Furthermore, we only considered bugs for which the original fixed version could properly execute for all test cases generated by the *OCaml* property-based testing tool *Quickcheck* (Claessen and Hughes, 2000). Proper execution was defined as the absence of errors or timeouts for any given input. This was necessary to ensure that the bugs

[7] Actually, the function is polymorphic, but the test case requires a more specialized type, which is what we get thanks to inlining.

were genuine and that any improvements observed in our evaluation were a result of our tool's impact, rather than external factors such as faulty test cases or unreliable program behavior. After applying these filters, we evaluated a set of 1,318 bugs.

Validating the Generated Patches

To validate the effectiveness of our tool in repairing bugs, we used *Quickcheck* to generate a random, large number of test cases. Moreover, we define properties to assert that the human-fixed program is "equivalent" to the repaired one. Thus, for each bug, we generated a set of patches and automatically instantiated a corresponding *Quickcheck* property. This is expressed according to the following template:

```
1 let%test_unit "testName" =
2     Quickcheck.test
3         [%quickcheck.generator: <input_signature>]
4       ~f:(fun args ->
5         [%test_eq: <output_signature>]
6         (Fix.functionToTest args) (Gen.functionToTest args))
```

To generate a property for the faulty program being repaired, we consider the faulty function's signature. The input part of the signature (line 3) is used to implement a generator for the input values that will be tested. The output part (line 5) is used to tell *Quickcheck* the type of the output values to compare. Line 6 represents the property that should be verified and means that the result of the original fixed program should be equal to the result of the patch being tested. The number of arguments needs to be adjusted according to the function being tested and, as such, `args` is modified to reflect that. By default, the generator produces 10,000 inputs. If the property holds, the patch is considered equivalent to the fixed version.

A bug is considered to be repaired if at least one of the generated patches produced the same output as the original fixed version for all input combinations generated by *Quickcheck*. By automating the instantiation of this property for every considered bug, we were able to accurately validate the effectiveness of our tool in repairing bugs. This approach also allowed us to provide quantitative metrics on the performance of our tool, such as the percentage of bugs repaired and the degree to which some bugs are partially fixed — Figures 5.3 and 5.4. This automated validation process is crucial given the amount of data at this stage. Furthermore, it also provides some insight into how it could be incorporated in real-world use cases. To the best of

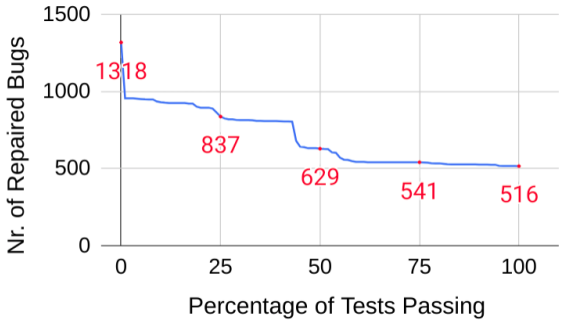


Figure 5.3: #Programs that pass at least a given percentage of tests. For example, 629 programs pass at least 50% of tests.

our knowledge, it is uncommon to provide a fully automatic validation process to verify whether generated patches successfully fix buggy programs. Our approach has the flexibility of allowing patches equivalent to the intended fix, without relying on human intervention to manually inspect the generated patches.

Results and Discussion

Our approach successfully repaired a substantial portion of the dataset. Among the $1{,}318$ bugs evaluated, our tool repaired 516 of them, achieving a repair rate of 39.2%. We found that 441 of the programs were partially fixed, indicating that the generated patches were able to address some but not all of the identified issues in the program, representing a 33.5% partial repair rate. The consideration of partial fixes provides a more nuanced understanding of the capabilities of our technique. Rather than simply categorizing a program as either fixed or not fixed, partial fixes enable us to explore the ground that separates a completely fixed program from a program that remains broken. Thus, we can form an idea of how the partial fixes are distributed along that spectrum. Out of the 361 programs that remained unfixed, we found that 247 of them produced some error during testing and the testing process did not finish. Additionally, 73 of the unfixed programs were due to our technique being unable to generate any patch for the identified bugs. Interestingly, we also found that 41 of the unfixed programs actually failed every test produced by *Quickcheck*, indicating that the bugs in these programs were particularly challenging to address.

Different tool modes exhibit varying degrees of repair effectiveness, as shown in Figure 5.5. The *Fill* strategy is the most effective, being responsible for fixing 394 out of the total 516 programs (76.4%). The *Instruct* strategy was also found to be effective, repairing 224 programs (43.4%). On the other hand, the *Choose* strategy is the least effective, with 108 fixed programs (20.9%).

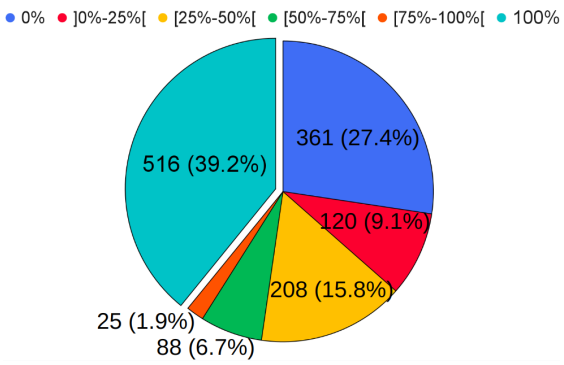


Figure 5.4: Distribution of test passing rate of programs. For example, 208 programs pass between 25% and 50% of tests.

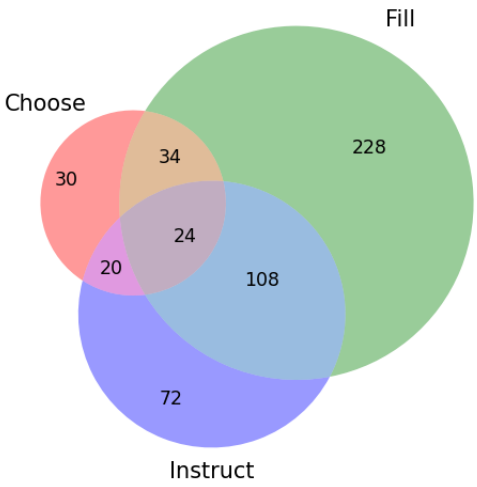


Figure 5.5: How many programs each mode successfully repairs. Intersections mean that a program is repaired correctly by both modes. There are 34 programs that can be repaired by both the Choose mode or the Fill mode.

It is worth noting that some programs were repaired by multiple strategies, and in some cases, the same program was repaired by all three strategies. Specifically, there were 108 (20.9%) programs that were fixed by both *Fill* and *Instruct*, while 34 (6.6%) programs were fixed by both *Fill* and *Choose*, and 20 (3.9%) programs were fixed by both *Choose* and *Instruct*. Additionally, there were 24 (4.7%) programs that were repaired by all three strategies.

Limitations

Our automated validation strategy excludes functions relying on user-defined data types, as it needs manually defined specific generators. This limitation reduces the number of programs we analyze, as discussed in Section 5.5.3. Moreover, we assume total functions, meaning that we consider every possible input for a given type, resulting in a more pessimistic repair validation. For instance, if a function has an integer as argument and is designed to work only with positive numbers, our fully automated approach will still test it with negative numbers (as produced by the predefined generator of integer numbers) reporting it as a non repaired function.[8]

Let us consider the *OCaml* implementation for *factorial*:

```
1 (* int -> int *)
2 let rec factorial n =
3   if n = 0 then 1
4   else n * factorial (n - 1)
```

The provided implementation is the usual recursive definition for *factorial*. Note this is a partial function as it is only defined for positive values of the input `n`. If `n` is a negative number, *factorial* will indefinitely call itself causing a stack overflow error. Now, let us consider that this implementation of *factorial* results from a repair process, either generated by Mentat or another tool. When we validate such repair with our automated validation approach, we use *Quickcheck* to automatically generate inputs for this function. In this case, the predefined generator for *int* will produce both positive and negative values. Although the repaired *factorial* function is correct, our validation will fail due to timeout as soon as it is called with a negative number.

[8] Generators for positive integers and user-defined types can be implemented. However, this would break our goal of a fully automated process.

5.5.4 Comparative Study

We performed a comparative study of our technique for automated program repair of ill-typed *OCaml* programs. We utilized the results provided in Rite's (Sakkas et al., 2020) repository for both their tool and Seminal to validate the efficacy of our fully automated validation strategy. In section 5.5.3, we acknowledge the demanding and pessimistic nature of our testing strategy by highlighting its consideration of total functions, encompassing every possible input for any given type. This ignores any restriction on the set of valid inputs. A manual validation process, similar to that employed by Rite, has the potential to increase the success rate for both our approach and the others. This kind of manual validation allows for a more sensitive consideration of program characteristics that may be overlooked by a more automated validation method. That is, some expected usage patterns may be better captured by a human evaluator with a more subjective evaluation criteria. An example is judging a function's implementation and considering it has been designed to only work with positive numbers, even though the type system may only reflect the function operates on type *int*. However, such manual validation would imply extensive manual effort and is infeasible for the size of this dataset.

We compare our approach with two other tools, namely Rite and Seminal, in terms of their repair capabilities on a common dataset. Although the three tools used a common dataset as an underlying basis, each work applied its own pre-processing criteria to prepare the dataset. As a consequence, in this comparative study, our original dataset of $1{,}318$ bugs was filtered down to 591 bugs, which were common to all three approaches. Figure 5.6 shows the distribution of the bugs and how they intersect among Mentat, Rite and Seminal.

Our technique achieved a repair rate of 37.6% (222 out of 591 programs). It employs a fully automated analysis that considers a program fixed only if it becomes well-typed and passes all test cases. Our repair process leverages GPT-3, a powerful large language model, to generate patches for identified type errors. This eliminates the need for a comprehensive system and language-specific components due to GPT-3's extensive training on multiple languages.

Originally, Rite conducted a manual validation through a user study with 29 programmers in which a set of 21 buggy programs was selected and each participant was shown 10 randomly selected buggy programs alongside two candidate repairs, one generated by Rite and one by Seminal. A full validation of the entire dataset was not reported. To achieve this, we used our

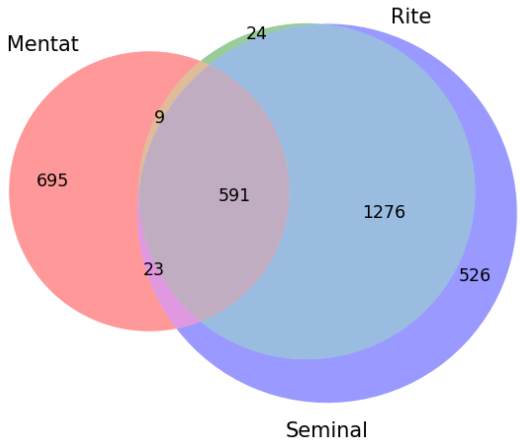


Figure 5.6: #Programs used in each repair technique and intersections.

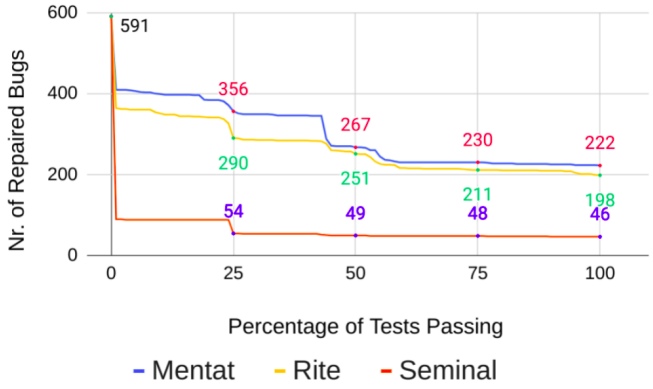


Figure 5.7: Number of programs that pass at least a given percentage of tests - comparative study.

automated validation framework to verify which Rite and Seminal generated patches were able to pass all test cases produced by *Quickcheck*.

This way, we were able to evaluate the performance of Rite and Seminal on the same dataset. Rite repaired 198 programs out of 591 (33.5% repair rate), while Seminal repaired only 46 programs (7.8% repair rate). These results highlight the superior effectiveness of our technique over the existing state-of-the-art tools for automated program repair in the context of type errors in *OCaml* programs. Figure 5.7 shows the repair effectiveness of the three tools.

One noteworthy advantage of our approach is its language-agnostic nature. Our technique can be easily adapted to repair programs in other languages, as long as it is possible to statically determine the types of terms either through inference or annotations, and the ability to bypass the type system exists (e.g., `Obj.magic` for *OCaml* or `undefined` for *Haskell*). Furthermore, the reliance on LLM's, such as GPT-3, for generating patches liberates us from building language-

specific generation systems for each case. By leveraging these prerequisites, our approach can be successfully applied to a wide range of programming languages.

We conclude that Mentat outperforms both Rite and Seminal in repairing type errors on a common dataset of *OCaml* programs. Our fully automated approach eliminates the need for user studies to validate patch relevance and ensures that the resulting programs are not only well-typed but also pass all the provided test cases.

Our results provide the following four insights: First, Mentat surpasses both Rite and Seminal in terms of effective program repair, i.e. patches are well-typed and are equivalent to the intended fixed version; Second, we thoroughly validated Rite's repairs, whereas their paper only validates 21 repairs with user involvement; Third, although Rite reports over 80% success in type repair, we show that the percentage of repairs passing the tests is 33.5%, which is significantly lower and highlights the potential for misleading results[9]; Fourth, our fully automated validation approach enabled us to validate other works that previously relied on manual analysis of a very limited subset of programs.

5.6 Related Work

Type error debugging research has a rich history spanning over 30 years, evolving from enhancing error messages (Damas and Milner, 1982; Wand, 1986; Lee and Yi, 1998) to interactive debugging tools (Chitil, 2001; Tsushima and Asai, 2013; Chen and Erwig, 2014a,b), and automated approaches that narrow down error causes (Haack and Wells, 2004; Rahli et al., 2017; Stuckey et al., 2003, 2004; Schilling, 2012). These methods aim to pinpoint errors and require user intervention for correction. On the other hand, automatic correction of type errors is a nascent field; Seminal (Lerner et al., 2007) is, to our knowledge, the first system for automatic correction of type errors in functional programming languages. It removes parts of the ill-typed program and attempts to make syntactic changes. This corresponds to *Fill* in our study: they used a syntactic modification to fill, and we used GPT-3. Rite (Sakkas et al., 2020) aims for program repair of ill-typed programs too. From a corpus of $4{,}500$ ill-typed *OCaml* programs, it uses approximately half of the dataset to build a neural network that learns what modifications have been made to, ultimately, synthesize solutions for given ill-typed programs. Our tool, Mentat performs

[9]This also contradicts the (informal) usual saying in functional programming: if it type checks, then it is correct.

source code analysis to produce useful prompts that leverage GPT-3's language understanding and generation capabilities to generate potential patches. While Mentat, Rite and Seminal share a common objective of fixing ill-typed *OCaml* programs, they diverge in their validation methodologies. Rite relies on a manual analysis of 21 randomly selected programs from the repository by a limited number of programmers, whereas our technique employs a fully automated process to validate the generated repairs. This distinction allows our technique to perform validation on a larger scale, effectively addressing the challenges associated with manual validation processes. The definition of a fixed program in Rite is based on the ability of the generated program to typecheck correctly. In contrast, our work validates both typechecking and semantical equivalence of the generated repairs. To achieve this, our technique employs a methodology that generates and executes test cases for both the correct program and the generated repairs. It considers a program to be fully repaired only if the correct program and the repaired version produce identical outputs for all test cases. This crucial difference allowed us to verify that a pure type repair can fall short of being an effective repair. We demonstrated that Rite's reported $+80\%$ type repair rate is comparatively lower in terms of actual program repair, i.e. the generated repair satisfies the test cases 33.5% of times. As we mentioned in Section 5.5.3, this is based on a pessimistic view that a patch must pass all test cases. Indeed, a manual analysis may reveal that more of the generated patches are semantically equivalent to the intended program, potentially improving our results as well as those of Rite and Seminal.

DeepTyper (Hellendoorn et al., 2018) enhances type information for compilation using deep learning in *Python* and *JavaScript*. However, it lacks program repair capabilities. Our work utilizes *OCaml*'s type inference for source code analysis and prompt preparation. DeepTyper could be beneficial when extending our approach to other programming languages.

Fault localization (Ang et al., 2017; Pearson et al., 2017) is an initial debugging step (Parnin and Orso, 2011). Various methods, including execution trace analysis (Campos et al., 2012), mutation testing (Moon et al., 2014), qualitative reasoning (Perez and Abreu, 2018), and semantic fault identification (Ribeiro et al., 2021), help narrow down suspicious code elements. Models like *code2vec* (Alon et al., 2019) have been trained to specifically detect security vulnerabilities (Coimbra et al., 2021). Our work concentrates on type errors and uses *OCaml*'s type inference to identify potentially responsible expressions by transforming them into different types.

APR is a prominent research field. Early approaches use genetic programming (Arcuri, 2011;

Le Goues et al., 2012b), while others employ constraint-based methods (Durieux and Monperrus, 2016; Nguyen et al., 2013; Xuan et al., 2017). Recent advancements incorporate machine learning and neural machine translation techniques (Chen et al., 2021; Li et al., 2020; Lutellier et al., 2020). However, translating buggy code to fixed code has limitations (Ding et al., 2021) and general-purpose models supporting code understanding and generation tasks (Feng et al., 2020; Lu et al., 2021; Svyatkovskiy et al., 2020; Alon et al., 2019) started being considered. GPT-2's code completion effectively fixes *Java* bugs (Ribeiro et al., 2022), and *Codex* has repaired *Python* and *Java* programs (Prenner et al., 2022). Our work stands out for targeting type errors in *OCaml*, which prevent program compilation, unlike other research focused on functional bugs.

5.7 Summary

This chapter introduced a method to automatically fix type errors in *OCaml* programs using GPT-3. We achieve this by analyzing and modifying the faulty source code to create prompts for GPT-3-based models.

We developed the Mentat tool, initially validating it with simple programs and variations of the *Dijkstra* algorithm. In large-scale experiments involving $1{,}318$ buggy programs, we achieved a 39% repair rate using a novel automated patch validation approach. In comparison with two other *OCaml* program repair tools, Mentat outperformed them, achieving a 37.6% repair rate on a shared dataset of 591 programs, while the other tools achieved rates of 33.5% and 7.8%, respectively.

Replication Package

The necessary resources to replicate this study are publicly available:

- **Tool:** `gitlab.com/FranciscoRibeiro/mentat`
- **Artifact (Ribeiro et al., 2023b):** `doi.org/10.6084/m9.figshare.23646903.v2`

Conclusions

6

Throughout this thesis, we have examined how the absence of integrated context in fault localization (FL) and automated program repair (APR) techniques hinders them, thus limiting their full effectiveness. Our discussions illuminated shortcomings within existing approaches and proposed techniques to address these issues. At the core of our efforts was the aspiration to incorporate context into these domains and create a synergy with LLMs. Ultimately, we were motivated to answer the question (cf. Section 1.5):

Main Question

How can we incorporate LLMs into APR by combining context from source code and FL?

This question prompted us to break down our efforts into four peripheral questions — outlined in Section 1.5. Each of these focused on different aspects, making the exploration of the overarching problem more approachable, understandable, and effective. As we conclude, we will summarize the work done in each peripheral question's corresponding chapters, highlight the key contributions of this thesis, and suggest future research directions.

6.1 Peripheral Questions

Research Question 1

Can we describe a program's evolution using mutation operators? ***(Chapter 3)***

At its core, this question delves into an exploration in which we aim to provide insights into the reasons behind program faults, transitioning the emphasis from "where" to "why." This shift was prompted as a step towards a more comprehensive understanding of a program's evolution. Leveraging the concept of mutation operators from mutation testing, we apply them differently by exploring the inference of these operators from AST transformations. This addresses limitations

posed by textual diffs. The AST highlights a program's structure and content, and enables a detailed understanding of changes through additions, deletions, updates, or node movements. This provides valuable information related to the source code's context. In contrast, detecting textual-level changes during a file's evolution primarily considers insertions and deletions.

Thus, we developed Morpheus, a tool linking source code changes to mutation operators. Our algorithm, involving partitioning and matching, infers mutation operators from AST transformations. This process is comparable to type inference systems, where transformations (expressions) seek association with mutation operators (types), relying on detected properties (type rules). Our results show that this is a sound approach, as we were able to infer mutation operators for 78% of the 1496 valid mutants in CodeDefenders and that 10% of these are higher-order mutants (Jia and Harman, 2009).

Research Question 2

Does the information about inferred mutations benefit APR? ***(Chapter 3)***

For this question, our primary concern centered around utilizing the semantics of a bug to guide the repair process. The previous question paved the way to translating bug-inducing changes into mutation operators. We now wish to inform a repair strategy about this. The implementation of this repair strategy comprises three key steps. First, interpreting the report by Morpheus to create components linking inferred mutations to their location in the source code. Second, isolating tree nodes representing source code elements in specific locations. Third, applying the opposed mutation operator to generate patches.

Naturally, we implemented a repair tool based on those steps. Additionally, we conducted an in-depth examination of various case studies drawn from real-world projects in Bugswarm and Defects4J. This analysis highlighted the advantages of our approach in the context of APR. Namely, it outperforms SFL by detecting mutants more efficiently and more effectively, especially in pinpointing faults inside a faulty line. As it does not rely on program execution traces, it can analyze a wider range of programs. Also, it ensures reachability across a program's history as it does not need failing tests. Finally, it provides finer granularity than typical SFL components.

Research Question 3

Can LLM-generated code evolve and fix faulty programs? ***(Chapter 4)***

This question emerged from recent advancements in the development of DL-based pre-trained language models designed for code understanding and code generation. These models are typically created without a specific downstream task in mind. Our proposal is based on harnessing these models' capabilities and use them for APR. A key insight was recognizing the complexity in generating fixed versions, irrespective of bug location awareness. These intricacies often demand prior knowledge, challenging traditional APR techniques relying on rigid patterns. LLMs, with their extensive code learning, offer more adaptability and understanding, capturing diverse coding styles and contextual cues. Our inspiration stems from the core concept of code completion, crafted to support developers during code development. In this context, developers place the cursor where they seek relevant completions. Hence, we approached the repair task as a code completion challenge, employing the assistance of CodeGPT, a code generation model.

To address this problem, we presented an automated repair technique that, given a buggy file and line number, generates candidate patch lines using a truncation algorithm. We devised a truncation algorithm that computes appropriate column numbers based on textual boundaries of language constructs and naming conventions. Additionally, we leverage CodeGPT for code completion in a multi-step process involving cutting, code generation, bounding, and character synchronization. This involves parsing the buggy code, generating multiple token sequences, constraining based on relevant syntax characters, and aligning characters for potential patches. Our approach, validated on the ManySStuBs4J dataset, achieved a 27% repair rate, fixing 1739 out of 6415 programs, which we consider affirms the efficacy and robustness of our methodology.

Research Question 4

Can integrating type system information into LLM interaction fix type errors? ***(Chapter 5)***

Let us first make an observation: current type systems, despite identifying type inconsistencies, often struggle to pinpoint the exact location and reasons behind these errors. This limitation fuels our motivation to explore a less conventional approach: seeking automated repairs as a means to not only fix type inconsistencies but also uncover and comprehend the underlying type errors. With this in mind, our OCaml program repair approach utilizes GPT-3 to address type inconsistencies. Leveraging a function we call typecast, we identify expressions causing type errors in a three-step process: Type Error Location, Inlining, and Type Unification. The foundation of our strategy lies in a specialized function, typecast, which manipulates the compiler's

interpretation of expressions. This enables us to bypass the type system, a critical aspect of our approach. Type error location marks problematic expressions, inlining associates function usage with context for the type system, and type unification filters completion suggestions for typed holes, unifying them with flagged expressions. Additionally, we automatically confirm the success of repairs by employing Quickcheck to validate that the generated patches yield identical output to the user-intended fixed version.

Our approach achieved a commendable 39.2% repair rate across 1,318 evaluated bugs. Out of the repaired programs, 516 were fully fixed, and 441 received partial fixes, providing insights into our technique's effectiveness. Analyzing programs for which some degree of repairability was achieved, but not full, enables a spectrum of insights, along with a more comprehensive understanding of the repair process beyond a binary categorization of fixed or unfixed. Moreover, among the 361 unfixed programs, 247 encountered errors during testing, 73 remained unfixed due to patch generation challenges, and 41 presented notably challenging issues, failing every test from Quickcheck.

In a comparative analysis with Rite and Seminal on a shared dataset, our approach achieved a superior repair rate of 37.6% (222 out of 591 programs), outperforming Rite (33.5%) and Seminal (7.8%). Notably, our method excelled in effective program repair, ensuring well-typed and equivalent patches. Additionally, we conducted thorough validations of Rite's repairs, revealing discrepancies in their reported success rates and emphasizing the potential for misleading outcomes. Our fully automated validation approach allowed us to scrutinize a broader range of programs compared to previous works reliant on manual analysis of limited subsets.

6.2 Summary of Contributions

The main contributions of this thesis are the following:

- An inference technique that translates source code changes to mutation operators. The approach, implemented as the Morpheus tool, demonstrates soundness by successfully inferring mutation operators for a significant portion of valid mutants. Additionally, the analysis extends to real-world projects, showcasing the benefits of our approach regarding APR.
- A program repair technique, treating the repair task as code completion. For a specified file

and line, it computes adequate column numbers and then uses CodeGPT to generate code. Effort is placed on restricting generated code sequences to maintain syntactic harmony with the original program, ensuring seamless integration. The validation phase on real-world open-source projects demonstrated the technique's effectiveness.

- A method, Mentat, utilizing GPT-3 to address OCaml type inconsistencies. A specialized function enables us to bypass the type system, with a multi-step process identifying, associating, and unifying ill-typed expressions. We use Quickcheck for fully automatic patch validation. Validation revealed the technique is effective and outperforms existing state of the art approaches.

6.3 Recommendations for Future Research

Building on the insights shared in this thesis, we consider there are several interesting questions that can be explored. These inquiries not only address existing limitations but also chart a course for continuing investigations. As such, we suggest some ideas for future research:

In Chapter 3, the objective was to diverge from SFL's emphasis on "where" faults occur, aiming instead to offer contextual insights into the "why" of fault occurrences. This shift provides a more comprehensive understanding of fault origins and underscores the significant impact that inferred information can have on the effectiveness of program repair. This path can be expanded in several ways:

- **Repair Granularity** Firstly, the application of multiple mutations, either through their compounding to construct more intricate expressions or their discrete application at separate locations to address independent modifications, presents a compelling direction. Additionally, considering broader contextual information for repair, with the aim of implementing different repair strategies, like those based on Q-SFL's landmarks, holds potential for more nuanced and effective patching.
- **Context Customization** Furthermore, delving into the inference of patterns beyond documented ones, potentially reporting unnamed patterns when detected, could enrich both the comprehensibility of fault localization and the repertoire of repair strategies. Lastly, the parameterization of repair techniques to receive specific operators to apply, opens up the possibility for more customizable and targeted repair approaches, which

can unlock certain scenarios and broaden the approach's impact.

Our primary focus in Chapter 4 was evaluating the effectiveness of directly applying LLMs for APR, treating it as a code completion task. To achieve this, we developed a truncation algorithm that calculates column numbers, leveraging CodeGPT to execute code completion.

- **Precise and Efficient Patch Generation** A pivotal aim is to enhance the precision and efficiency of generated patches. To achieve this, focus should shift towards minimizing the number of syntactically incorrect patches through a meticulous consideration of structural aspects within the program. Specifically, delving into the examination of AST node types in the generated code becomes imperative, ensuring seamless integration of expressions into the original program and thereby filtering out erroneous patches.
- **Sampling Impact** Additionally, an exploration into the impact of different sampling techniques on code generation, such as top-k and nucleus sampling, is warranted. Given the observed disparities in probability distributions between decoded text and human-written text, as highlighted by (Holtzman et al., 2020), investigating the generalizability of these findings to programming languages emerges as an intriguing avenue for further exploration.

The work in Chapter 5 introduced an approach that leverages the capabilities of GPT-3 models to automatically address type errors in OCaml programs. The method synergistically analyzed source code and harnessed the power of type systems to generate targeted prompts, contributing to automated resolutions of OCaml type errors.

- **Authentic Scenario Coverage** Our evaluation primarily focused on the analysis of real-world programs created by students for a functional programming course. However, we acknowledge that, despite being representative of a real-world scenario—learning the language, going through the course exercises — effectively capturing the reported difficulties students face when dealing with type errors, these programs may not entirely encompass the complexity of production-ready applications. As such, a pragmatic expansion of our approach involves the application of our type error repair methodology to real-world open-source projects in OCaml. Given the intricate nature of collecting and setting up these projects, which demands significant time and resources, addressing this practical challenge could bolster the applicability of APR methodologies in authentic development scenarios.

- **Reproducibility** At the time of the work in Chapter 5, achieving fully reproducible results was challenging due to the inherent non-deterministic nature of completions through OpenAI's API. This resulted in potential variations in model outputs from request to request. Despite these challenges, we meticulously addressed all available concerns, ensuring consistency in parameters such as prompt, top-p, and temperature across requests. Notably, OpenAI has since introduced enhanced control over deterministic outputs by providing access to the seed parameter and the system_fingerprint response field.
- **Multiple Models** Additionally, considering local, open-source models would also offer more control over experiments while allowing for extensive comparisons between LLMs. Certainly, conducting this form of experimentation demands powerful computing resources, which were not readily available to us at the time. However, overcoming these challenges opens the door to harnessing the numerous models that have been recently released and continue to emerge in the field. Furthermore, beyond generative models, there are several alternatives worth exploring, such as infilling models like CodeBERT and InCoder, instruction-following models like Alpaca and the more recent mixture of experts Mixtral 8X7B.
- **Dynamically-Typed Languages** The challenge posed by addressing type errors in dynamically-typed languages, like Python, is rooted in their flexible nature. Unlike statically-typed languages, where variable types are explicitly declared, dynamically-typed languages allow variables to change types during runtime. This flexibility enhances expressiveness but introduces ambiguity when errors arise, making it more intricate to pinpoint and resolve. Additionally, Python's reliance on runtime checks and the absence of a strict type-enforcement mechanism can complicate the identification and correction of type-related issues, adding another layer of complexity to the debugging process. Therefore, exploring effective strategies for handling the dynamic context of languages like Python becomes imperative for robust program repair.
- **Multi-Error Type Issues** Type systems exhibit limitations in addressing complex scenarios where multiple errors contribute to type-related issues. In some cases, a single type error may have a cascading effect, triggering subsequent errors or inconsistencies in different parts of the code. Consequently, resolving one type error might not be sufficient to fully rectify the program, as other related errors persist, leaving the overall system still ill-typed. This arises due to the interconnected nature of type dependencies within a

program. Fixing one type error might reveal latent errors elsewhere, requiring a holistic approach to navigate and resolve the web of dependencies. The inherent challenge lies in the fact that type systems often operate on a per-error basis, potentially overlooking the broader context of interconnected errors that collectively contribute to the ill-typed nature of the program. Therefore, addressing multiple fixes in the context of type errors demands a nuanced exploration that goes beyond isolated solutions.

- **Context Specific** Moreover, the fine-tuning of models for specific repair contexts, such as type errors, arithmetic operations, or even for specific languages like OCaml, presents a tailored and potentially more effective approach. However, this specialization may come at the cost of losing some level of generalizability. Therefore, the justification for such targeted measures must be thoroughly assessed, ensuring that the nature of the repair scenarios warrants the trade-off between specificity and broader applicability. Examples would be domain-specific application with unique coding patterns or constraints or language-specific nuances with certain languages possessing distinctive nuances in expressivity, which may give rise to unique error patterns and idiosyncrasies. Lastly, it is crucial to note that the pursuit for such highly effective systems may inadvertently lead to the common pitfall of overfitting, which besides compromising adaptability also results in zero effectiveness.

References

7

Annual "humies" awards for human-competitive results, 2004. URL https://human-competitive.org/. Accessed in 2023-10-31.

Research by cambridge mbas for tech firm undo finds software bugs cost the industry $316 billion a year, 2013. URL https://www.jbs.cam.ac.uk/2013/research-by-cambridge-mbas-for-tech-firm-undo-finds-software-bugs-cost-the-industry-316-billion-a-year/. Accessed in 2023-12-19.

The error that could subvert george osborne's austerity programme, 2013. URL https://www.theguardian.com/politics/2013/apr/18/uncovered-error-george-osborne-austerity. Accessed in 2023-12-20.

Mozilla's security bug bounty program, 2017. URL https://www.mozilla.org/en-US/security/bug-bounty/. Accessed in 2023-12-19.

Welcome to google's bug hunting community, 2021. URL https://bughunters.google.com/. Accessed in 2023-12-19.

Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46, 2006. 10.1109/PRDC.2006.18.

Rui Abreu, Wolfgang Mayer, Markus Stumptner, and Arjan J. C. van Gemund. Refining spectrum-based fault localization rankings. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, page 409–414, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605581668. 10.1145/1529282.1529374. URL https://doi.org/10.1145/1529282.1529374.

Sérgio Almeida, Ana C. R. Paiva, and André Restivo. Mutation-based web test case generation. In Mario Piattini, Paulo Rupino da Cunha, Ignacio García Rodríguez de Guzmán, and Ricardo Pérez-Castillo, editors, *Quality of Information and Communications Technology*, pages 339–346, Cham, 2019. Springer International Publishing. ISBN 978-3-030-29238-6.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. 10.1145/3290353. URL https://doi.org/10.1145/3290353.

J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 402–411, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139632. 10.1145/1062455.1062530. URL https://doi.org/10.1145/1062455.1062530.

James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Trans. Softw. Eng.*, 32 (8):608–624, aug 2006. ISSN 0098-5589. 10.1109/TSE.2006.83. URL https://doi.org/10.1109/TSE.2006.83.

Aaron Ang, Alexandre Perez, Arie Van Deursen, and Rui Abreu. Revisiting the practical use of automated software fault localization techniques. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 175–182, 2017. 10.1109/ISSREW.2017.68.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology EXchange*, eclipse '05, page 35–39, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595933425. 10.1145/1117696.1117704. URL https://doi.org/10.1145/1117696.1117704.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 361–370, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. 10.1145/1134285.1134336. URL https://doi.org/10.1145/1134285.1134336.

Andrea Arcuri. Evolutionary repair of faulty software. *Appl. Soft Comput.*, 11(4):3494–3514, jun 2011. ISSN 1568-4946. 10.1016/j.asoc.2011.01.023. URL https://doi.org/10.1016/j.asoc.2011.01.023.

Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on*

Foundations of Software Engineering, FSE 2014, page 306–317, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. 10.1145/2635868.2635898. URL https://doi.org/10.1145/2635868.2635898.

Earl T. Barr, Mark Harman, Yue Jia, Alexandru Marginean, and Justyna Petke. Automated software transplantation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 257–269, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. 10.1145/2771783.2771796. URL https://doi.org/10.1145/2771783.2771796.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc. ISBN 9781713829546.

Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 213–222, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605580012. 10.1145/1595696.1595728. URL https://doi.org/10.1145/1595696.1595728.

José Campos, André Riboira, Alexandre Perez, and Rui Abreu. Gzoltar: An eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE '12, page 378–381, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312042. 10.1145/2351676.2351752. URL https://doi.org/10.1145/2351676.2351752.

Saikat Chakraborty, Yangruibo Ding, Miltiadis Allamanis, and Baishakhi Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, 48(4):1385–1399, 2022. 10.1109/TSE.2020.3020502.

Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, page 121–130, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304450. 10.1145/1985793.1985811. URL https://doi.org/10.1145/1985793.1985811.

Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, page 493–504, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917944. 10.1145/233269.233366. URL https://doi.org/10.1145/233269.233366.

M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604, 2002. 10.1109/DSN.2002.1029005.

Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In *Symposium on Principles of Programming Languages. Proceedings*, POPL '14, pages 583–594. ACM, 2014a. ISBN 9781450325448. 10.1145/2535838.2535863. URL https://doi.org/10.1145/2535838.2535863.

Sheng Chen and Martin Erwig. Guided type debugging. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming. Proceedings*, LNCS 8475, pages 35–51. Springer, 2014b. 10.1007/978-3-319-07151-0_3. URL https://doi.org/10.1007/978-3-319-07151-0_3.

Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering*, 47(9):1943–1959, 2021. 10.1109/TSE.2019.2940179.

Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *International Conference on Functional Programming. Proceedings*, ICFP '01, pages 193–204. ACM, 2001. ISBN 1581134150. 10.1145/507635.507659. URL https://doi.org/10.1145/507635.507659.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

David Coimbra, Sofia Reis, Rui Abreu, Corina Pasareanu, and Hakan Erdogmus. On using distributed representations of source code for the detection of C security vulnerabilities. *arXiv e-prints*, art. arXiv:2106.01367, June 2021. 10.48550/arXiv.2106.01367.

Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: a practical mutation testing tool for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 449–452, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. 10.1145/2931037.2948707. URL https://doi.org/10.1145/2931037.2948707.

Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, OOPSLA '08, page 313–328, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582153. 10.1145/1449764.1449790. URL https://doi.org/10.1145/1449764.1449790.

Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, page 528–550, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 354027992X. 10.1007/11531142_23. URL https://doi.org/10.1007/11531142_23.

Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Symposium on Principles of Programming Languages. Proceedings*, POPL '82, pages 207–212. ACM, 1982. ISBN 0897910656. 10.1145/582153.582176. URL https://doi.org/10.1145/582153.582176.

Murial Daran and Pascale Thévenod-Fosse. Software error analysis: A real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, page 158–171, New York, NY, USA, 1996.

Association for Computing Machinery. ISBN 0897917871. 10.1145/229000.226313. URL https://doi.org/10.1145/229000.226313.

Johan de Kleer. Multiple representations of knowledge in a mechanics problem-solver. In Daniel S. Weld and Johan de Kleer, editors, *Readings in Qualitative Reasoning About Physical Systems*, pages 40–45. Morgan Kaufmann, 1990. ISBN 978-1-4832-1447-4. https://doi.org/10.1016/B978-1-4832-1447-4.50009-2. URL https://www.sciencedirect.com/science/article/pii/B9781483214474500092.

Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. ISSN 0004-3702. https://doi.org/10.1016/0004-3702(87)90063-4. URL https://www.sciencedirect.com/science/article/pii/0004370287900634.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, page 65–74, USA, 2010. IEEE Computer Society. ISBN 9780769539904. 10.1109/ICST.2010.66. URL https://doi.org/10.1109/ICST.2010.66.

Vidroha Debroy and W. Eric Wong. Combining mutation and fault localization for automated program debugging. *J. Syst. Softw.*, 90(C):45–60, apr 2014. ISSN 0164-1212. 10.1016/j.jss.2013.10.042. URL https://doi.org/10.1016/j.jss.2013.10.042.

Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, page 30–39, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328470. 10.1145/2593735.2593740. URL https://doi.org/10.1145/2593735.2593740.

R. A. DeMillo and Aditya Mathur. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical report, 01 1990.

R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, apr 1978. ISSN 0018-9162. 10.1109/C-M.1978.218136. URL https://doi.org/10.1109/C-M.1978.218136.

Anna Derezin□ska. Advanced mutation operators applicable in c# programs. In Krzysztof Sacha, editor, *Software Engineering Techniques: Design for Quality*, pages 283–288, Boston, MA, 2007. Springer US. ISBN 978-0-387-39388-9.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. 10.18653/v1/N19-1423. URL https://aclanthology.org/N19-1423.

E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271, 1959. ISSN 0029-599X. 10.1007/BF01386390.

Yangruibo Ding, Baishakhi Ray, Premkumar Devanbu, and Vincent J. Hellendoorn. Patching as translation: the data and the metaphor. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, page 275–286, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. 10.1145/3324884.3416587. URL https://doi.org/10.1145/3324884.3416587.

Thomas Durieux and Martin Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, page 85–91, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341516. 10.1145/2896921.2896931. URL https://doi.org/10.1145/2896921.2896931.

Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE

2019, page 302–313, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450355728. 10.1145/3338906.3338911. URL https://doi.org/10.1145/3338906.3338911.

Rafael Dutra, Rahul Gopinath, and Andreas Zeller. Formatfuzzer: Effective fuzzing of binary file formats. *ACM Trans. Softw. Eng. Methodol.*, 33(2), dec 2023. ISSN 1049-331X. 10.1145/3628157. URL https://doi.org/10.1145/3628157.

Martin Eberlein, Marius Smytzek, Dominic Steinhöfel, Lars Grunske, and Andreas Zeller. Semantic debugging. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, page 438–449, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703270. 10.1145/3611643.3616296. URL https://doi.org/10.1145/3611643.3616296.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, page 57–72, New York, NY, USA, 2001. Association for Computing Machinery. ISBN 1581133898. 10.1145/502034.502041. URL https://doi.org/10.1145/502034.502041.

Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, may 2000. ISSN 1520-9202. 10.1109/6294.846201. URL https://doi.org/10.1109/6294.846201.

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. 10.1145/2642937.2642982. URL https://doi.org/10.1145/2642937.2642982.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In Trevor Cohn, Yulan He, and Yang Liu, editors, *Findings of the*

Association for Computational Linguistics: EMNLP 2020, pages 1536–1547, Online, November 2020. Association for Computational Linguistics. 10.18653/v1/2020.findings-emnlp.139. URL https://aclanthology.org/2020.findings-emnlp.139.

Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, 33(11):725–743, nov 2007. ISSN 0098-5589. 10.1109/TSE.2007.70731. URL https://doi.org/10.1109/TSE.2007.70731.

Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 328–343, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. 10.1145/3064176.3064183. URL https://doi.org/10.1145/3064176.3064183.

Kenneth D Forbus. *Qualitative reasoning*, pages 62–1–62–19. CRC Press, January 2004. ISBN 9781584883609. Publisher Copyright: © 2004 by CRC Press, LLC.

Philip L. Frana and Thomas J. Misa. An interview with edsger w. dijkstra. *Commun. ACM*, 53(8):41–47, aug 2010. ISSN 0001-0782. 10.1145/1787234.1787249. URL https://doi.org/10.1145/1787234.1787249.

Gordon Fraser and Andrea 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*, ESEC/FSE '11, page 416–419, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450304436. 10.1145/2025113.2025179. URL https://doi.org/10.1145/2025113.2025179.

Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 147–158, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781605588230. 10.1145/1831708.1831728. URL https://doi.org/10.1145/1831708.1831728.

Matthías Páll Gissurarson, Leonhard Applis, Annibale Panichella, Arie van Deursen, and David Sands. Propr: Property-based automatic program repair. In *Proceedings of the*

44th International Conference on Software Engineering, ICSE '22, page 1768–1780, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. 10.1145/3510003.3510620. URL https://doi.org/10.1145/3510003.3510620.

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

Christian Haack and J.B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1):189–224, 2004. ISSN 0167-6423. https://doi.org/10.1016/j.scico.2004.01.004. URL https://www.sciencedirect.com/science/article/pii/S016764230400005X. 12th European Symposium on Programming (ESOP 2003).

B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Syst. J.*, 41 (1):4–12, jan 2002. ISSN 0018-8670. 10.1147/sj.411.0004. URL https://doi.org/10.1147/sj.411.0004.

Quinn Hanam, Fernando S. de M. Brito, and Ali Mesbah. Discovering bug patterns in javascript. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 144–156, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342186. 10.1145/2950290.2950308. URL https://doi.org/10.1145/2950290.2950308.

Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. *SIGPLAN Not.*, 33(7):83–90, jul 1998. ISSN 0362-1340. 10.1145/277633.277647. URL https://doi.org/10.1145/277633.277647.

Hideaki Hata, Emad Shihab, and Graham Neubig. Learning to Generate Corrective Patches using Neural Machine Translation. *arXiv e-prints*, art. arXiv:1812.07170, December 2018. 10.48550/arXiv.1812.07170.

Bastiaan Heeren, Johan Jeuring, S. Swierstra, and Pablo Azero Alcocer. Improving type-error messages in functional languages. 03 2002.

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 62–71,

New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137583. 10.1145/871895.871902. URL https://doi.org/10.1145/871895.871902.

Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 152–162, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355735. 10.1145/3236024.3236051. URL https://doi.org/10.1145/3236024.3236051.

Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, August 2012. USENIX Association. ISBN 978-931971-95-9. URL https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

Shin Hong, Byeongcheol Lee, Taehoon Kwak, Yiru Jeon, Bongsuk Ko, Yunho Kim, and Moonzoo Kim. Mutation-based fault localization for real-world multilingual programs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, page 464–475. IEEE Press, 2015. ISBN 9781509000241. 10.1109/ASE.2015.14. URL https://doi.org/10.1109/ASE.2015.14.

Yue Jia and Mark Harman. Higher order mutation testing. *Inf. Softw. Technol.*, 51(10): 1379–1393, oct 2009. ISSN 0950-5849. 10.1016/j.infsof.2009.04.016. URL https://doi.org/10.1016/j.infsof.2009.04.016.

Nan Jiang, Thibaud Lutellier, and Lin Tan. Cure: Code-aware neural machine translation for automatic program repair. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 1161–1173. IEEE Press, 2021. ISBN 9781450390859. 10.1109/ICSE43902.2021.00107. URL https://doi.org/10.1109/ICSE43902.2021.00107.

James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, page 467–477, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 158113472X. 10.1145/581339.581397. URL https://doi.org/10.1145/581339.581397.

René Just. The major mutation framework: efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 433–436, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450326452. 10.1145/2610384.2628053. URL https://doi.org/10.1145/2610384.2628053.

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, page 654–665, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330565. 10.1145/2635868.2635929. URL https://doi.org/10.1145/2635868.2635929.

Rafael-Michael Karampatsis and Charles Sutton. *How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset*, page 573–577. Association for Computing Machinery, New York, NY, USA, 2020. ISBN 9781450375177. URL https://doi.org/10.1145/3379597.3387491.

Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 802–811. IEEE Press, 2013. ISBN 9781467330763.

Sunwoo Kim, John A Clark, and John A McDermid. Class mutation: Mutation testing for object-oriented programs. In *Proceedings of the Net. Object Days Conference on Object-Oriented Software Systems*, pages 9–12, 2000.

K. N. King and A. Jefferson Offutt. A fortran language system for mutation-based software testing. *Softw. Pract. Exper.*, 21(7):685–718, jun 1991. ISSN 0038-0644. 10.1002/spe.4380210704. URL https://doi.org/10.1002/spe.4380210704.

Johannes Lampel, Sascha Just, Sven Apel, and Andreas Zeller. When life gives you oranges: detecting and diagnosing intermittent job failures at mozilla. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, page 1381–1392, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. 10.1145/3468264.3473931. URL `https://doi.org/10.1145/3468264.3473931`.

Xuan Bach D. Le, David Lo, and Claire Le Goues. History driven program repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224, 2016. 10.1109/SANER.2016.76.

Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 3–13. IEEE Press, 2012a. ISBN 9781467310673.

Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, jan 2012b. ISSN 0098-5589. 10.1109/TSE.2011.104. URL `https://doi.org/10.1109/TSE.2011.104`.

Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998. ISSN 0164-0925. 10.1145/291891.291892. URL `https://doi.org/10.1145/291891.291892`.

Benjamin S. Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. *SIGPLAN Not.*, 42(6):425–434, jun 2007. ISSN 0362-1340. 10.1145/1273442.1250783. URL `https://doi.org/10.1145/1273442.1250783`.

Yi Li, Shaohua Wang, and Tien N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *ICSE '20*, 2020. ISBN 9781450371216. 10.1145/3377811.3380345. URL `https://doi.org/10.1145/3377811.3380345`.

Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable

statistical bug isolation. *SIGPLAN Not.*, 40(6):15–26, jun 2005. ISSN 0362-1340. 10.1145/1064978.1065014. URL https://doi.org/10.1145/1064978.1065014.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 298–312, New York, NY, USA, 2016a. Association for Computing Machinery. ISBN 9781450335492. 10.1145/2837614.2837617. URL https://doi.org/10.1145/2837614.2837617.

Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 702–713, New York, NY, USA, 2016b. Association for Computing Machinery. ISBN 9781450339001. 10.1145/2884781.2884872. URL https://doi.org/10.1145/2884781.2884872.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. CodeXGLUE: A machine learning benchmark dataset for code understanding and generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021. URL https://openreview.net/forum?id=6lE4dQXaUcb.

Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *ISSTA 2020*, 2020. ISBN 9781450380089. 10.1145/3395363.3397369. URL https://doi.org/10.1145/3395363.3397369.

Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, page 352, USA, 2002. IEEE Computer Society. ISBN 0818617633.

Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava: a mutation system for java. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 827–830,

New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. 10.1145/1134285.1134425. URL https://doi.org/10.1145/1134285.1134425.

Christian Macho, Shane Mcintosh, and Martin Pinzger. Extracting build changes with builddiff. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*. IEEE Press, 2017.

Fernanda Madeiral, Thomas Durieux, Victor Sobreira, and Marcelo Maia. Towards an automated approach for bug fix pattern detection. *arXiv e-prints*, art. arXiv:1807.11286, July 2018. 10.48550/arXiv.1807.11286.

Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. 10.1145/2931037.2931054. URL https://doi.org/10.1145/2931037.2931054.

Matias Martinez and Martin Monperrus. Astor: a program repair library for java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 441–444, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450343909. 10.1145/2931037.2948705. URL https://doi.org/10.1145/2931037.2948705.

Matias Martinez, Westley Weimer, and Martin Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, page 492–495, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327688. 10.1145/2591062.2591114. URL https://doi.org/10.1145/2591062.2591114.

Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. *Empirical Softw. Engg.*, 22(4):1936–1964, aug 2017. ISSN 1382-3256. 10.1007/s10664-016-9470-4. URL https://doi.org/10.1007/s10664-016-9470-4.

Ehsan Mashhadi and Hadi Hemmati. Applying codebert for automated program repair of java simple bugs. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 505–509, 2021. 10.1109/MSR52588.2021.00063.

Wolfgang Mayer, Rui Abreu, Markus Stumptner, Arjan JC Van Gemund, et al. Prioritising model-based debugging diagnostic reports. In *Proceedings of the 19th International Workshop on Principles of Diagnosis*, pages 127–134. Citeseer, 2008.

Andréia da Silva Meyer, Antonio Augusto Franco Garcia, Anete Pereira de Souza, and Cláudio Lopes de Souza Jr. Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l). *Genetics and Molecular Biology*, 27(1):83–91, 2004. ISSN 1415-4757. 10.1590/S1415-47572004000100014. URL `https://doi.org/10.1590/S1415-47572004000100014`.

Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the mutants: Mutating faulty programs for fault localization. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation*, ICST '14, page 153–162, USA, 2014. IEEE Computer Society. ISBN 9781479922550. 10.1109/ICST.2014.28. URL `https://doi.org/10.1109/ICST.2014.28`.

Madan Musuvathi and Dawson R Engler. Some lessons from using static analysis and software model checking for bug finding. *Electronic Notes in Theoretical Computer Science*, 89(3), 2003.

Akbar Siami Namin and Sahitya Kakarla. The use of mutation in testing experiments and its sensitivity to external threats. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 342–352, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305624. 10.1145/2001420.2001461. URL `https://doi.org/10.1145/2001420.2001461`.

Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3), feb 2016. ISSN 0360-0300. 10.1145/2841425. URL `https://doi.org/10.1145/2841425`.

Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Pro-

gram repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 772–781. IEEE Press, 2013. ISBN 9781467330763.

A. Offutt. The coupling effect: Fact or fiction. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, TAV3, page 131–140, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913426. 10.1145/75308.75324. URL https://doi.org/10.1145/75308.75324.

A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.*, 1(1):5–20, jan 1992. ISSN 1049-331X. 10.1145/125489.125473. URL https://doi.org/10.1145/125489.125473.

Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. The class-level mutants of mujava. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, page 78–84, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595934081. 10.1145/1138929.1138945. URL https://doi.org/10.1145/1138929.1138945.

Mike Papadakis and Yves Le Traon. Using mutants to locate "unknown" faults. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST'12)*. IEEE Press, 2012.

Mike Papadakis and Yves Le Traon. Metallaxis-fl: Mutation-based fault localization. *Softw. Test. Verif. Reliab.*, 25(5–7):605–628, aug 2015. ISSN 0960-0833. 10.1002/stvr.1509. URL https://doi.org/10.1002/stvr.1509.

Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 199–209, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305624. 10.1145/2001420.2001445. URL https://doi.org/10.1145/2001420.2001445.

Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *ICSE '17*, 2017. ISBN 9781538638682. 10.1109/ICSE.2017.62. URL https://doi.org/10.1109/ICSE.2017.62.

Alexandre Perez and Rui Abreu. Leveraging qualitative reasoning to improve sfl. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, IJCAI'18, page 1935–1941. AAAI Press, 2018. ISBN 9780999241127.

Alexandre Perez, Rui Abreu, and Marcelo D'Amorim. Prevalence of single-fault fixes and its impact on fault localization. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 12–22, 2017. 10.1109/ICST.2017.9.

Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002. ISBN 0262162091.

Macario Polo, Mario Piattini, and Ignacio García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing, Verification and Reliability*, 19(2):111–131, 2009. https://doi.org/10.1002/stvr.392. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.392`.

Julian Aron Prenner, Hlib Babii, and Romain Robbes. Can openai's codex fix bugs? an evaluation on quixbugs. In *Proceedings of the Third International Workshop on Automated Program Repair*, pages 69–75, 2022.

Zichao Qi, Fan Long, Sara Achour, and Martin 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*, ISSTA 2015, page 24–36, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336208. 10.1145/2771783.2771791. URL `https://doi.org/10.1145/2771783.2771791`.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. Skalpel: A constraint-based type error slicer for standard ml. *Journal of Symbolic Computation*, 80:164–208, 2017. ISSN 0747-7171. https://doi.org/10.1016/j.jsc.2016.07.013. URL `https://www.sciencedirect.com/science/article/pii/S0747717116300505`. SI: Program Verification.

C. V. Ramamoorthy and Wei-Tek Tsai. Advances in software engineering. *Computer*, 29(10): 47–58, oct 1996. ISSN 0018-9162. 10.1109/2.539720. URL https://doi.org/10.1109/2.539720.

U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. Di Guglielmo, G. Pravadelli, and F. Fummi. Combining dynamic slicing and mutation operators for esl correction. In *2012 17th IEEE European Test Symposium (ETS)*, 2012.

Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22(6):432–449, nov 1997. ISSN 0163-5948. 10.1145/267896.267925. URL https://doi.org/10.1145/267896.267925.

Francisco Ribeiro, Rui Abreu, and João Saraiva. On understanding contextual changes of failures. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 1036–1047, 2021. 10.1109/QRS54544.2021.00112.

Francisco Ribeiro, Rui Abreu, and João Saraiva. Framing program repair as code completion. In *Proceedings of the Third International Workshop on Automated Program Repair*, APR '22, page 38–45, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392853. 10.1145/3524459.3527347. URL https://doi.org/10.1145/3524459.3527347.

Francisco Ribeiro, José Nuno Castro de Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. Gpt-3-powered type error debugging: Investigating the use of large language models for code repair. In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2023, page 111–124, New York, NY, USA, 2023a. Association for Computing Machinery. ISBN 9798400703966. 10.1145/3623476.3623522. URL https://doi.org/10.1145/3623476.3623522.

Francisco Ribeiro, José Macedo, Kanae Tsushima, Rui Abreu, and João Saraiva. GPT-3-Powered Type Error Debugging: Investigating the Use of Large Language Models for Code Repair (replication artifact), 10 2023b. URL https://figshare.com/articles/software/GPT-3-Powered_Type_Error_Debugging_Investigating_the_Use_of_Large_Language_Models_for_Code_Repair_SLE_2023_/23646903.

José Miguel Rojas, Thomas D. White, Benjamin S. Clegg, and Gordon Fraser. Code defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. IEEE Press, 2017.

Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *FM 2016: Formal Methods*, pages 593–611, Cham, 2016. Springer International Publishing. ISBN 978-3-319-48989-6.

Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. Reconstructing core dumps. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 114–123, 2013. 10.1109/ICST.2013.18.

Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 16–30, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. 10.1145/3385412.3386005. URL `https://doi.org/10.1145/3385412.3386005`.

Thomas Schilling. Constraint-free type error slicing. In Ricardo Peña and Rex Page, editors, *Trends in Functional Programming. Proceedings*, LNCS 7193, pages 1–16. Springer, 2012. ISBN 978-3-642-32037-8. 10.1007/978-3-642-32037-8_1.

Robert C. seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003. ISBN 0321118847.

Ridwan Salihin Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. Automated patch transplantation. *ACM Trans. Softw. Eng. Methodol.*, 30(1), dec 2021. ISSN 1049-331X. 10.1145/3412376. URL `https://doi.org/10.1145/3412376`.

Brian Cantwell Smith. The limits of correctness. *SIGCAS Comput. Soc.*, 14,15(1,2,3,4):18–26, jan 1985. ISSN 0095-2737. 10.1145/379486.379512. URL `https://doi.org/10.1145/379486.379512`.

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th*

Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, page 532–543, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336758. 10.1145/2786805.2786825. URL https://doi.org/10.1145/2786805.2786825.

Dominic Steinhöfel and Andreas Zeller. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. 10.1145/3540250.3549139. URL https://doi.org/10.1145/3540250.3549139.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Interactive type debugging in haskell. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, Haskell '03, page 72–83, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137583. 10.1145/871895.871903. URL https://doi.org/10.1145/871895.871903.

Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. Improving type error diagnosis. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 80–91, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138504. 10.1145/1017472.1017486. URL https://doi.org/10.1145/1017472.1017486.

Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: code generation using transformer. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 1433–1443, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. 10.1145/3368089.3417058. URL https://doi.org/10.1145/3368089.3417058.

Shin Hwei Tan and Abhik Roychoudhury. Relifix: Automated repair of software regressions. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*. IEEE Press, 2015.

Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology*, 7007(11):1–309, 2002. URL https://cir.nii.ac.jp/crid/1574231875579089536.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications, 2022.

Yuan Tian, David Lo, and Chengnian Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, page 200–209, USA, 2013. IEEE Computer Society. ISBN 9780769549811. 10.1109/ICSM.2013.31. URL `https://doi.org/10.1109/ICSM.2013.31`.

David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019.

Kanae Tsushima and Kenichi Asai. An embedded type debugger. In Ralf Hinze, editor, *Implementation and Application of Functional Languages*, pages 190–206, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-41582-1.

Kanae Tsushima and Kenichi Asai. A weighted type-error slicer. *Journal of Computer Software*, 31(4):131–148, 2014.

Kanae Tsushima, Olaf Chitil, and Joanna Sharrad. Type debugging with counter-factual type error messages using an existing type checker. In *Symposium on Implementation and Application of Functional Languages. Proceedings*, IFL '19. ACM, 2019. ISBN 9781450375627. 10.1145/3412932.3412939. URL `https://doi.org/10.1145/3412932.3412939`.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys

Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, page 832–837, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450359375. 10.1145/3238147.3240732. URL https://doi.org/10.1145/3238147.3240732.

Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 25–36. IEEE Press, 2019a. 10.1109/ICSE.2019.00021. URL https://doi.org/10.1109/ICSE.2019.00021.

Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4), sep 2019b. ISSN 1049-331X. 10.1145/3340544. URL https://doi.org/10.1145/3340544.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985. ISSN 0020-7373. https://-doi.org/10.1016/S0020-7373(85)80054-7. URL https://www.sciencedirect.com/science/article/pii/S0020737385800547.

Mitchell Wand. Finding the source of type errors. In *Symposium on Principles of Programming Languages. Proceedings*, POPL '86, pages 38–43. ACM, 1986. ISBN 9781450373470. 10.1145/512644.512648. URL https://doi.org/10.1145/512644.512648.

Jiawei Wang, Li Li, and Andreas Zeller. Restoring execution environments of jupyter notebooks.

In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1622–1633, 2021. 10.1109/ICSE43902.2021.00144.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *Transactions on Machine Learning Research*, 2022. ISSN 2835-8856. URL https://openreview.net/forum?id=yzkSU5zdwD. Survey Certification.

Brian C. Williams and Johan de Kleer. Qualitative reasoning about physical systems: A return to roots. *Artif. Intell.*, 51(1–3):1–9, oct 1991. ISSN 0004-3702. 10.1016/0004-3702(91)90106-T. URL https://doi.org/10.1016/0004-3702(91)90106-T.

W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016. 10.1109/TSE.2016.2521368.

Franz Wotawa, Markus Stumptner, and Wolfgang Mayer. Model-based debugging or how to diagnose programs automatically. In Tim Hendtlass and Moonis Ali, editors, *Developments in Applied Artificial Intelligence*, pages 746–757, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. ISBN 978-3-540-48035-8.

Chunqiu Steven Xia and Lingming 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*, ESEC/FSE 2022, page 959–971, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. 10.1145/3540250.3549101. URL https://doi.org/10.1145/3540250.3549101.

Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs. *IEEE Trans. Softw. Eng.*, 43(1):34–55, jan 2017. ISSN 0098-5589. 10.1109/TSE.2016.2560811. URL https://doi.org/10.1109/TSE.2016.2560811.

Aoyang Yan, Hao Zhong, Daohan Song, and Li Jia. How do programmers fix bugs as workarounds? an empirical study on apache projects. *Empirical Softw. Engg.*, 28(4), jun 2023. ISSN 1382-3256. 10.1007/s10664-023-10318-7. URL https://doi.org/10.1007/s10664-023-10318-7.

He Ye, Matias Martinez, and Martin Monperrus. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1506–1518, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392211. 10.1145/3510003.3510222. URL https://doi.org/10.1145/3510003.3510222.

Andreas Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. Injecting mechanical faults to localize developer faults for evolving software. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA'13)*. ACM, 2013.

Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, page 272–281, New York, NY, USA, 2006. Association for Computing Machinery. ISBN 1595933751. 10.1145/1134285.1134324. URL https://doi.org/10.1145/1134285.1134324.

Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu 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*, ESEC/FSE 2021, page 341–353, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. 10.1145/3468264.3468544. URL https://doi.org/10.1145/3468264.3468544.

Qihao Zhu, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. Tare: Type-aware neural program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1443–1455, 2023. 10.1109/ICSE48619.2023.00126.