FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

# Empirical Study on Live Automatic Program Repair

**Amadeu Prazeres Pereira**

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Restivo, Assistant Professor

Co-Supervisor: Hugo Ferreira, Assistant Professor

July 19, 2021

# Empirical Study on Live Automatic Program Repair

**Amadeu Prazeres Pereira**

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Tiago Boldt Sousa

External Examiner: Prof. João Saraiva

Supervisor: Prof. André Restivo

July 19, 2021

# Abstract

Software development is becoming increasingly complex. Assuring a project's quality is becoming harder since the time and resources needed to achieve quality software and find and fix bugs are also growing rapidly. To automate and simplify the development process, tools have been developed to improve the confidence in the product's quality and accelerate the development process since developers are given immediate feedback for their actions.

To help developers, tools like pAPRika were created. This tool leverages unit tests and property-based tests as specifications to generate variations in the source code to repair bugs in JavaScript and TypeScript. With this, pAPRika can provide live semantic feedback to the developers while they are writing code.

The necessity for tools like this arose because tools capable of providing feedback to their users frequently only provide syntactic information and rarely give semantic ones.

However, pAPRika faces the problem that every time a code segment is changed, it runs the whole test suite. This hinders its scalability to more significant projects. A way to solve this is by running only the tests that cover the modified code segments. With this in mind, we proposed to implement a fault localization technique that uses reverse code coverage.

Our solution was implemented on top of the existing pAPRika tool, and an empirical study was conducted to validate it. The study consisted of 10 participants and showed that the fault localization technique significantly reduces the number of tests needed to run while demonstrating that they were necessary to test the program.

**Keywords**: Automatic Program Repair (APR), Automatic Programming (AP), Live Automatic Programming Repair (LAPR), Fault Localization (FL), Reverse-Code-Coverage (RCC)

# Resumo

O desenvolvimento de software está a tornar-se cada vez mais complexo. Assegurar a qualidade de um projeto está a tornar-se mais difícil porque o tempo e os recursos necessários para atingir software de qualidade e encontrar e corrigir *bugs* também está a crescer rápidamente. Para automatizar e simplificar o processo de desenvolvimento, ferramentas estão a ser desenvolvidas para melhorar a confiança na qualidade do produto e acelerar o processo de desenvolvimento porque os desenvolvedores recebem *feedback* imediatamente pelas suas ações.

Para ajudar os desenvolvedores, ferramentas como pAPRika foram criadas. Esta ferramenta usa testes unitários e testes baseados em propriedades para gerar variações no código fonte para reparar *bugs* em JavaScript e TypeScript. Com isto, pAPRika pode fornecer *feedback* semântico em tempo real para os desenvolvedores enquanto estão a escrever código.

A necessidade para ferramentas como esta surgiu porque as ferramentas capazes de fornecer *feedback* para os seus utilizadores frequentemente só dão informação sintática e raramente dão informação semântica.

Contudo, pAPRika sofre o problemas de que sempre que um segmento de código é alterado, toda a suíte de testes é corrida. Isto afeta a sua escalabilidade para projetos maiores. Uma forma de resolver isto é correndo só os teste que cobrem o segmento de código modificado. Com isto em mente, propomos a implementação de uma técnica de localização de erros que usa cobertura de código reverso.

A nossa solução foi implementado por cima da já existente ferramenta pAPRika, e um estudo empírico foi conduzido para valida-la. Este estudo consisitiu em 10 participantes e mostrou que a técnica localização de erros reduz significativamente os testes necessários correr enquanto demonstra que esses testas são necessários para testar o programa.

**Keywords**: Automatic Program Repair (APR), Automatic Programming (AP), Live Automatic Programming Repair (LAPR), Fault Localization (FL), Reverse-Code-Coverage (RCC)

# Acknowledgements

First of all, I want to express my gratitude towards my supervisors, Professors André Restivo and Hugo Sereno, for guiding me throughout this work. Thank you for providing me with fundamental knowledge that allowed me accomplish a better work.

Lots of love to my parents, for the constant support, help and unconditional love. I am sincerely grateful for of you. Diogo and Maria, you both know how much each one of you means to me, thank you for being who you are.

Thank you for everything. I am who I am because of you.

I want to finish by thanking all my friends, from Viseu and Porto, for being such a massive part of my life. Thank you for the developed friendships in the last 5 years to some, thank you for much longer to others. Thank you for all the laughter, all the disagreements, and all the good memories that I have with all of you. Thank you for all the support.

Amadeu Pereira

*“Problems are not stop signs,*
*they are guidelines.”*

Robert H. Schiuller

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| AP | Automatic Programming |
| APR | Automatic Program Repair |
| AST | Abstract Syntax Tree |
| FL | Fault Localization |
| G&V | Generate-and-Validate |
| IDE | Integrated Development Environment |
| LAPR | Live Automatic Programming Repair |
| LSP | Language Server Protocol |
| RCC | Reverse Code Coverage |
| RQ | Research Questions |

# Chapter 1

# Introduction

This chapter introduces the problem under study. Section 1.1 presents the context of the work followed by its motivation in Section 1.2. Then, Section 1.3 details the problem under investigation, followed by Section 1.4 describes the main goals intended to achieve. Finally, Section 1.5 presents the structure of this document.

## 1.1 Context

Software development is becoming inherently complex, which is directly related to the increasing difficulty of debugging software. This debugging process costs companies billions of dollars annually because it is needed to allocate developers' time to fix, find and repair software bugs manually, consuming around 50% of their time [8].

This problem motivated researchers to investigate ways to reduce the software development cycle by automating its steps. One of the areas that emerged from these researches is Automatic Program Repair (APR), a field of study based on Automatic Programming (AP), a term first introduced in, at least, 1954 [7]. AP consists on the automatic generation of a computer program, converting high-level specification into an efficient implementation, whereas APR aims to reduce the manual effort required to debug a program. Automatic Program Repair receives a faulty program and its correct specification as input, automatically identifying a failure, implementing a fix, and validating that fix, ensuring that the program's specifications are followed, and unwanted behaviors were introduced [28, 12].

On top of this, there are other research areas that also aim to provide developers a way to to build software programs, guaranteeing its software quality. One of those areas is Live Programming (LP) [38], which entails the notion of giving live feedback to the users. LP aims at reducing the usual three-phased edit-compile-run software development cycle (or edit-compile-link-run) to just a one-phased one, where all the efforts are directed to the *edit* part.

## 1.2 Motivation

Software debugging is an essential but time-consuming process in software development, which is becoming increasingly complex [8]. This increase in complexity has a direct impact on the costs associated with debugging. As such, there is a great motivation to localize a fault in a program and patch it automatically.

Given this problem, APR techniques have been used to automatically debug and fix programs, abstracting all of its complexity from developers.

Merging Automatic Program Repair techniques with Live Programming ones in developing environments, could help reduce the effort put in debugging by fixing a bug while the software is still being developed. This could be achieved by providing tactile *semantic* feedback to the developers, which focus goes to developing source code, about the state of the program. This allows us to achieve a higher software development rate with a decrease in its associated cost, assuring that the project's specifications are met.

## 1.3 Problem

Current software development environments mainly only provide syntactic suggestions to the developers' writing code, giving no semantic information. The majority of APR implementations that have been developed fail to integrate with well-known development environments. This problem is inherently related to the lack of live *semantic* suggestions presented by APR tools.

Furthermore, tools that present this behavior — provide live suggestions to developers — are specific to a single development environment, to a single programming language, or lack providing relevant feedback in a reasonable time in an autonomous way.

In this work, we intend to improve an existing APR tool [10] in the form of a Visual Studio Code extension, to enhance its functionalities, performance, and usability by introducing new capabilities. More specifically, reduce the time required to find a faulty code segment and improve its general usability. The problem under study is further analyzed in Chapter 4.

## 1.4 General Goal

The main goal of this work is to extend a live automatic program repair tool created by Campos [9, 10, 36]. This tool, created as an extension to the Visual Studio Code development environment, provides live *semantic* feedback capable of generating program variations for faulty functions and validate them against the given test-specification.

Our extension of the tool should (1) implement a fault localization technique capable of detecting which subset of the test suite should be run, and (2) automate all its processes, allowing the tool to not depend on the user to start its repair process.

## 1.5 Document Structure

This document presents seven chapters, structured as follows:

- Chapter 1, **Introduction**, presents the problem under study, its goals and motivation;
- Chapter 2, **Background**, introduces the key concepts required to fully understand this work;
- Chapter 3, **State of the Art**, describes the current state of the art on the topic of automatic program repair;
- Chapter 4, **Problem Statement**, details the problem under study, the proposed solution on how to solve it, and its validation process;
- Chapter 5, **Proposed Solution**, delves into the implementation details of our solution, providing a justification to all the adopted choices and exploring all of its limitations;
- Chapter 6, **Empirical Study**, presents our validation process, analyzing all the obtained results;
- Chapter 7, **Conclusions**, presents a summary of the work done, describing its main contributions and future work.

# Chapter 2

# Background

This chapter introduces and describes key concepts needed in the further understanding of this work. Section 2.1 presents the definition and main goals of Automatic Program Repair as well as its types of specifications in Section 2.1.1. Section 2.1.2 introduces APR techniques, and in Section 2.1.3, an analysis of current APR limitations. An overview of what is Live Programming is presented in Section 2.2. Finally, this chapter's contents are summarized in Section 2.3

## 2.1 Automatic Program Repair

Automatic Program Repair (APR) is a research area whose main idea is automatically repairing a faulty software program by generating code changes (patches) that fix it [12]. This document follows the definition that *"Automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification"* [27].

As input, APR tools receive a program with some failure and a specification of its proper behavior.

There are two different types of patches: (1) *plausible* patches, ensure the program's correctness, following its specification, and (2) *correct* patches, fix the program without introducing unwanted behaviors [46].

Most APR approaches follow a two step methodology:

- **Fault Localization (FL)**: consists of localizing possible bugs in faulty software's source code. *Wong et al.* [43] classified FL techniques into eight categories: slice-based, spectrum-based, statistics-based, program state-based, machine learning-based, data mining-based, model-based and miscellaneous techniques. With spectrum-based fault localization (SBFL) implementations [16, 14, 2] being the most adopted ones.

- **Patch Generation**: uses patch ingredients to search for modified programs [20], which represent plausible patches, and their corresponding validation. These techniques will be explained in Section 2.1.2.

### 2.1.1 Specification

APR's goal is to find variations of a faulty program that fix it. A specification defining its correct behavior must be provided. This allows (1) identifying if the program is behaving incorrectly, and (2) provides a way to validate the suggested patches. These specifications serve as a guide that informs APR tools if the generated bug fixes satisfy the program's intended features.

There are several ways to defined a program's correct behavior. The most common ones used in APR are *formal-specifications*, *test-specifications*.

Formal-specifications provide an accurate and reliable way of specifying a program's wanted behavior. But, since they are normally human-written, they tend to be rarely available and error-prone [18, 22]. On the other hand, there are test-specifications, which are based on test suites. These are easier to establish but usually lack to define some behaviors, not providing a complete program specification [30]. If a plausible patch passes all given tests, it is assumed as a correct one but there is no way of ensuring that an unwanted behavior was not introduced [46].

### 2.1.2 Patch Generation Techniques

One of the steps required to generate automatic program fixes is *patch generation*. This tries to patch a bug by identifying code changes that may present a fix to the program. Existing repair techniques usually fall into two categories: *generate-and-validate (G&V)* and *semantics-based*. These will be explained in detail in Section 2.1.2.1 and Section 2.1.2.2, respectively.

#### 2.1.2.1 Generate-and-Validate

Generate-and-validate techniques follow the process described in Figure 2.1. As the name suggests, it is divided into two steps: (1) *generate*, where candidate patches to the original program are created, and (2) *validate*, where such fixes' correctness is verified.

The candidate patches generation is commonly done using three different *change operators*:

- **Atomic change:** alters the faulty program in just one location. This can be achieved by changing the Abstract Syntax Tree (AST), by inserting, removing, modifying statements or expressions or reusing them from elsewhere in the source code.

- **Pre-defined templates:** using templates allows altering one or more locations of the program's source code, being able to fix more complex bugs. These templates are retrieved by the APR tool's developers and present guidelines on how to fix given bugs [23, 25, 32].

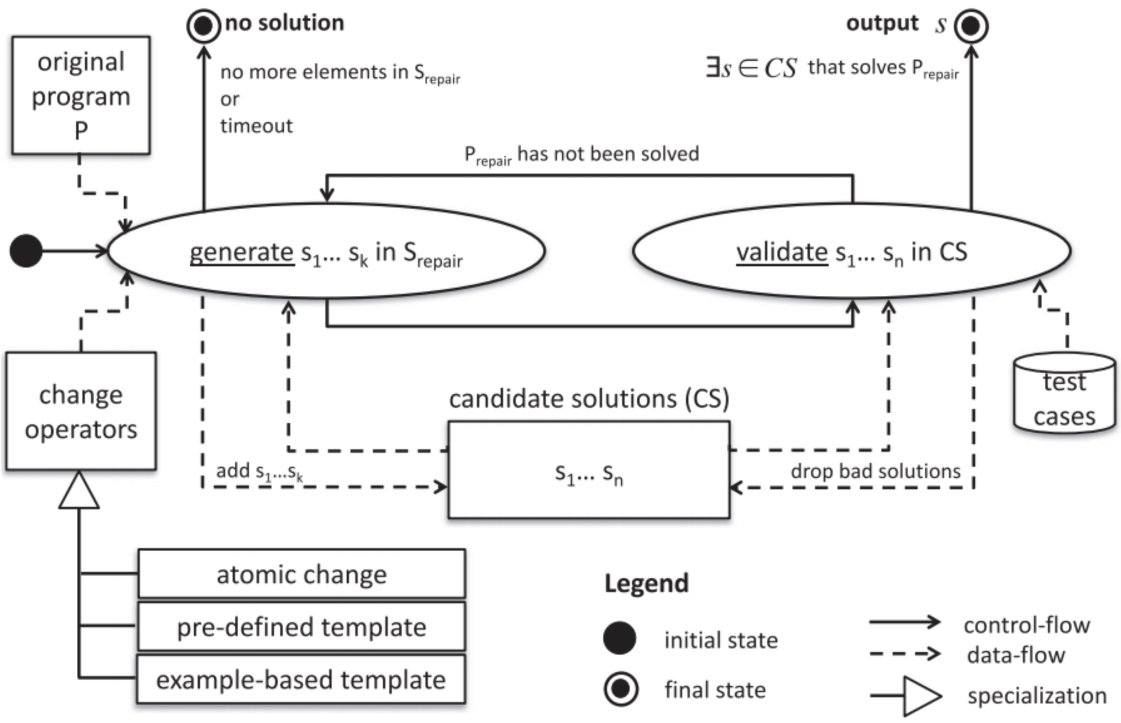

Figure 2.1: Generate-and-validate repair process [12].

- **Example-based templates:** works in a similar way to the previous one, but the templates are obtained by reusing previous fixes to the same problem [44, 15, 19].

In the validation step, the candidate patches' correctness is assessed by using the provided program's specification.

### 2.1.2.2 Semantics-based

Semantics-based techniques follow the process described in Figure 2.2. These approaches use symbolic execution and test cases to gather semantic information about a faulty code segment, which is then encoded as a formula [11]. This formula's solutions correspond to possible patches to the faulty program. These patches are then generated by program synthesis, and there is no need to validate them since they present the desired functionality, guaranteeing to fix the problem [30].

This technique consists of three main stages [12]:

- ***Behavioral analysis*:** In this step the program under repair is analyzed and semantic information about its correct and faulty behaviors is extracted.
- ***Problem generation*:** Uses the information retrieved in the previous step to create a formal representation of the problem. The solutions to this problems represent the patches the fix the faulty program.

- ***Fix generation*:** Tries to solve the problem generated. Identifies the code change that fixes the program. This stage either finds a valid solution, returning the repaired program, or return no solution if it does not exist or can't be found in a reasonable time.

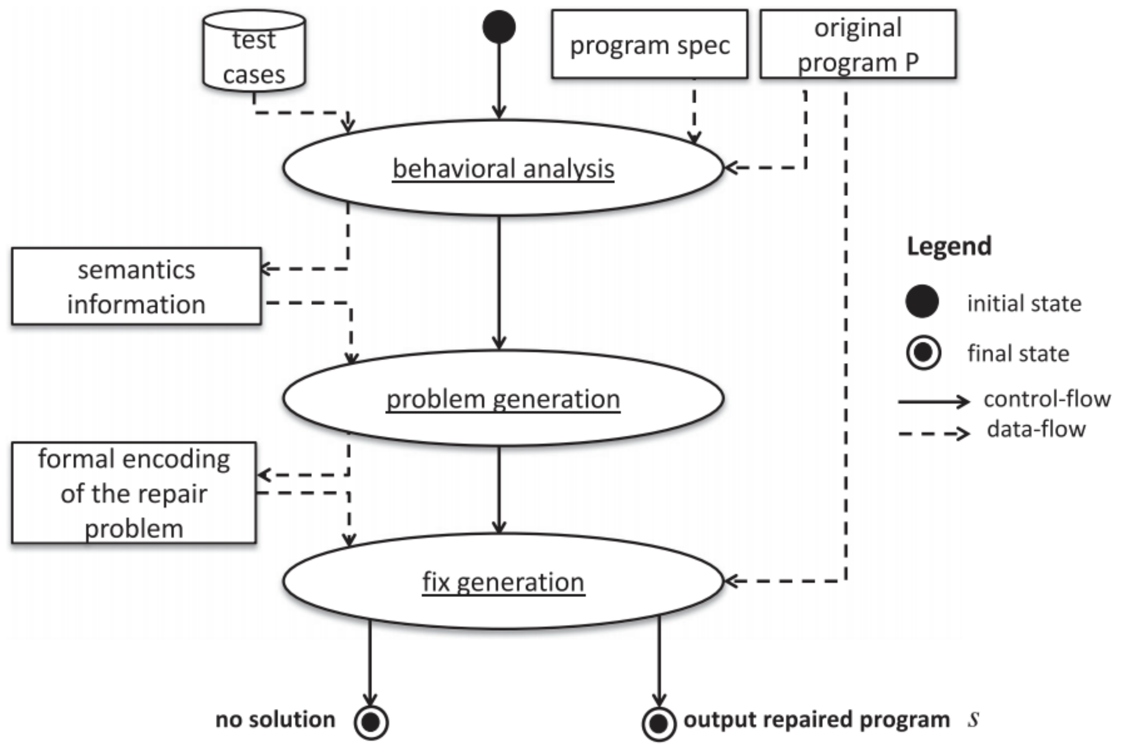

Figure 2.2: Semantics-based repair process [12].

### 2.1.3 Limitations

Current APR tools mainly suffer from two main limitations that prevent them from being widely used:

- **Search space explosion:** APR tools aim to find a program variation that is capable of fixing a bug in the space of all possible variations (*search space*). *Long and Rinard* [24] conducted a study that shows: (1) a considerable search space could be insufficient to find a correct patch, and (2) the search space contains a lot of plausible but incorrect patches that hinder the find of the correct one. A small search space might not include the correct patches, with the solution being its expansion, hindering the repair performance.

- **Patch overfitting:** Current repair techniques often *overfit* patched programs. These programs successfully follow the given program's specification but fail to properly fix the bug due to an incomplete specification.

## 2.2 Live Programming

Live Programming, like APR, aims to fasten the software development process. It is a research area built around the notion of providing live feedback to developers [3].

When programming, the developer follows a three-phased edit-compile-run software development cycle (or edit-compile-link-run). When live programming, this development cycle is reduced to only one phase (*edit*). Instead of manually running the typical three phases, the programmer only has to worry about writing source code since the developing environment automatically runs the other phases while providing live feedback [38].

The ability of an environment to automatically provide feedback to programmers about what they are developing is defined as **liveness** [38]. This characteristic can be measured in a six level hierarchy [27], as shown in Figure 2.3.

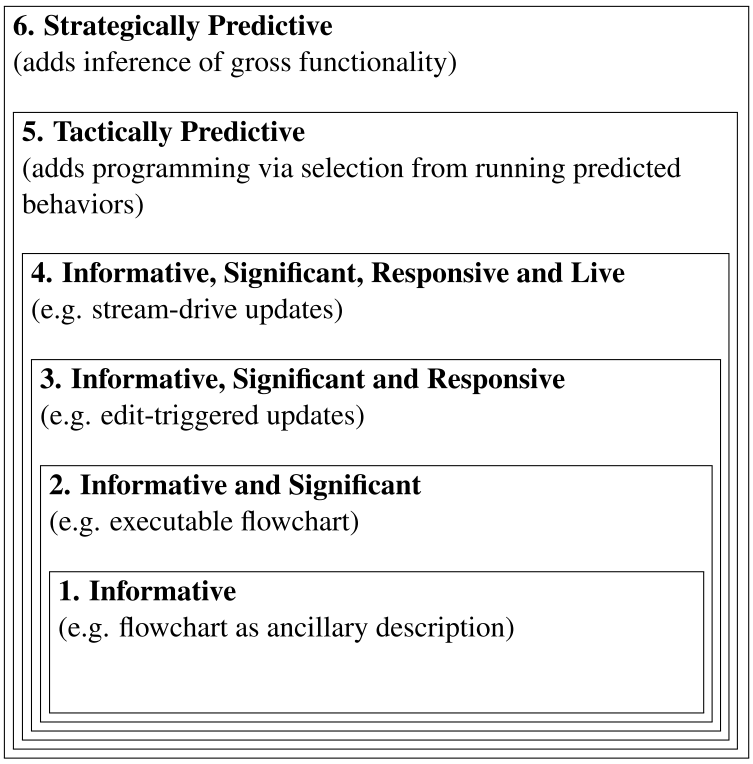
6. Strategically Predictive
(adds inference of gross functionality)
5. Tactically Predictive
(adds programming via selection from running predicted behaviors)
4. Informative, Significant, Responsive and Live
(e.g. stream-drive updates)
3. Informative, Significant and Responsive
(e.g. edit-triggered updates)
2. Informative and Significant
(e.g. executable flowchart)
1. Informative
(e.g. flowchart as ancillary description)

Figure 2.3: Liveness hierarchy proposed by Tanimoto [38].

## 2.3 Summary

This chapter presents concepts related to Automatic Program Repair, and Live Programming which are essential to understand this work. Section 2.1 defines APR an presents several techniques that are associated with this research area. These techniques are then analyzed in detail, outlining the problems that they currently face.

Section 2.2 presents the concept of Live Programming, introducing the concept of *liveness* and depicting how it is measured.

# Chapter 3

# State of the Art

This chapter describes the state of the art of automated program repair approaches and how studies are conducted on these same approaches. Section 3.1 presents the conducted Systematic Literature Review to gather relevant information for this state of the art, describing the used methodology in Section 3.1.1, which aims to answer the survey research questions defined in Section 3.1.1.1. Section 3.1.2 presents the results of the Systematic Literature Review, which are categorized in Section 3.1.3. Section 3.1.4 presents the analysis and discussion of the approaches found, answering the previously defined survey questions in Section 3.1.4.2. Lastly, Section 3.2 contains a summary of the main findings on the mentioned topics.

## 3.1 Systematic Literature Review

A systematic literature review (SLR) was made to review the state of the art of empirical studies applied to APR methods or tools. The goal of conducting this type of review is to gather and synthesize enough results assuring their quality.

### 3.1.1 Methodology

This SLR followed a specific methodology aiming to reduce bias and gather the best results. Hence, research questions to be answered were defined as well as literature data sources. Then, the results were analyzed following a specified inclusion and exclusion criteria.

#### 3.1.1.1 Survey Research Questions

To identify the current state of the art approaches, techniques, and studies related to automatic program repair implementations, the following Survey Research Questions (SRQ) were established:

**SRQ1** *Which are the most relevant APR approaches?* APR tools follow different approaches to generate a fix to faulty program. Identifying these approaches is essential to understand how program repair tools work.

**SRQ2** *How is the performance of an APR approach evaluated?* Different approaches present different results. It is needed to identify how these results are evaluated to understand better how an approach's performance is measured.

**SRQ3** *How are APR tools validated?* When an APR tool is developed, it needs to be validated. It is necessary to understand how these validations are done and their corresponding validity threats.

**SRQ4** *Do APR tools provide live fixes to developers?* The liveness of an APR tool is an important characteristic [3] that is needed to be evaluated.

Answering these questions will provide valuable insight to practitioners on evaluating APR techniques and researchers by showing the current challenges and problems regarding this topic.

#### 3.1.1.2 Data Sources

The literature analyzed during this research was collected from three digital libraries: (1) Scopus, (2) ACM Digital Library, and (3) IEEE Xplore.

These electronic databases contain some of the most relevant literature for Computer Science studies, so they are considered reliable information sources.

#### 3.1.1.3 Literature Analysis

It is needed to select unbiased literature representative of the current state of the art, allowing the investigation of the previously defined research questions. Therefore, this SLR follows a set of inclusion and exclusion criteria detailed in Table 3.1.

Table 3.1: Inclusion and Exclusion Criteria.

| | **Inclusion Criteria** |
|---|---|
| IC1 | Must be on the topic of automatic programming or automatic program. |
| IC2 | Must belong to the Computer Science subject area. |
| IC3 | Published between 2011 and 2021. |
| IC4 | Provides a detailed empirical study about automatic program repair tools and/or methods. |
| | **Exclusion Criteria** |
| EC1 | Not written in English. |
| EC2 | Presents just ideas, integration experimentation, interviews, or discussion papers. |
| EC3 | Duplicate articles. |
| EC4 | Has less than five (non-self) citations when more than five years old. |
| EC5 | Not available in PDF. |
| EC6 | Presents a study not related to automatic program repair techniques. |

The literature review started with a general query to the digital databases to reach a significant amount of literature related to Automatic Program Repair.

```
("automat* program* repair" OR apr)
```

This way, we use a wildcard (⋆) to match different spelling variations of the same topic (e.g., automatic and automating, program and programming).

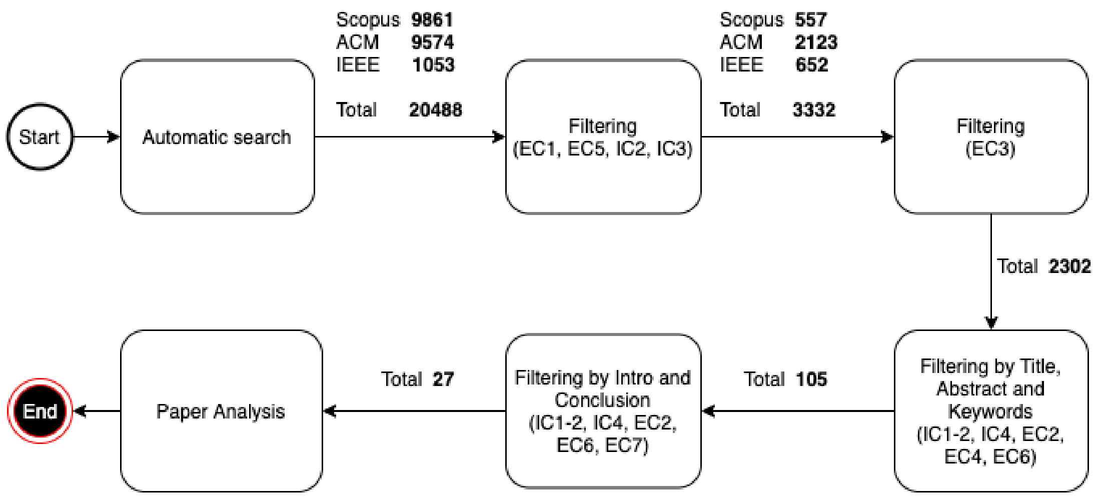
Start
Automatic search
Scopus 9861
ACM 9574
IEEE 1053
Total 20488
Filtering (EC1, EC5, IC2, IC3)
Scopus 557
ACM 2123
IEEE 652
Total 3332
Filtering (EC3)
Total 2302
Filtering by Title, Abstract and Keywords (IC1-2, IC4, EC2, EC4, EC6)
Total 105
Filtering by Intro and Conclusion (IC1-2, IC4, EC2, EC6, EC7)
Total 27
Paper Analysis
End

Figure 3.1: SLR Pipeline.

This review was conducted in January 2021 and is outlined in Figure 3.1. It presents a total of six stages with the following purposes:

1. **Automatic search:** Result gathered from the search in the digital libraries with the query mentioned above.

2. **Filtering (EC1 EC5 IC2 IC3):** The results obtained from the previous step were filtered by (EC1) language, filtering out the ones not written in English, (EC5) PDF availability, (IC2) subject area, eliminating the ones that don't belong to Computer Science, and (IC3) publish date, keeping only publications from 2011 to 2021.

3. **Filtering (EC3):** After merging all the papers from the different data sources, the duplicate entries were removed.

4. **Filtering by *Title*, *Abstract*, and *Keywords* (IC1-2 IC4 EC2 EC4 EC6)**: The resulting literature was reviewed according to its *Title*, *Abstract*, and *Keywords*. In this stage, papers presenting only ideas or hypotheses (EC2), having less than five (non-self) citations when more than five years old (EC4), a study not related to an Automatic Program Repair technique or tool (EC6), or not providing an evaluation on the proposed methods (IC4) were excluded.

5. **Filtering by *Introduction* and *Conclusion* (IC1-2 IC4 EC2 EC6 EC7):** Same procedure as the previous step, but now the literature analysis was expanded to the *Introduction* and *Conclusions sections*.
6. **Paper Analysis:** Detailed analysis of each publication to fully understand the presented study.

We started with a total of 20488 publications, but we narrowed them to 27 publications after a well defined systematic review protocol, as shown in Figure 3.1.

### 3.1.2 Results

**Xin and Reiss** [44] developed sharpFix when they noticed that ssFix [45] techniques could be significantly improved. ssFix is a search-based technique APR which repair process follows three steps: (1) *fault localization*, (2) *code search*, and (3) *code reuse*. sharpFix follows the same repair idea differing on how it approaches the code search and code reuse stages. These were suffering from using a basic code search approach for *local search* (within the local program) and *global search* (from a repository) and a not effective code reuse approach. The authors decided to improve the ssFix using two different *local search* methods (within the local program) and *global search* while adding one more step to the code reuse stage. The proposed tool methods were validated through two different experiments using the Defects4J dataset [17] and Bugs.jar-ELIXIR [37]. The results show that sharpFix outperformed ssFix in the first benchmark. For the second one, sharpFix was tested against ssFix and four other APR techniques, having the best performance of them all.

**Wang et al.** [40] present ARepair, a generate-and-validate APR technique for faulty Alloy models. It uses both mutation testing and program synthesis to generate a fix to the model. The authors defend that using a mixed approach is needed to find a complete fix for faulty models. An evaluation on the developed technique was conducted using human-written buggy Alloy models as input to ARepair (collected from graduate students' homework) and models from Alloy release 4.1, Amalgam [29]. To note that, the human-written models used are limited by the fact that most of them were written by graduate students, so the results can be *overfitted* and may not generalize to models written by skilled developers.

**Wen et al.** [42] introduce CapGen, a generate-and-validate APR that leverages fine-granularity fixing ingredients (e.g., expressions instead of a statement) to generate a correct patch. This tool aims to tackle the *space explosion* problem, prioritizing mutation operator and fix ingredients based on the likelihood of providing a correct fix instead of just a plausible one. The authors found that the context information of a suspicious segment of code and the given ingredients can give a good approximation about their likelihood to patch the program correctly, defending that "a fixing ingredient should be applied to the locations with similar contexts compared with the location where it is extracted." CapGen was evaluated on the Defects4J dataset against other APR techniques, achieving a precision of 84% (correctly fixed 21 bugs and generated 25 plausible fixes).

**Jiajun et al.** [15] present SimFix, an automatic program repair technique, which leverages existing patches from other projects and similar code snippets in the same project to generate patches. It follows two stages: *mining* and *repairing*. In the first one, repair patterns are mined from existing open-source projects, creating an *abstract space*. In the repairing stage, the tool extracts modifications from similar code snippets in the same program, constituting the *concrete space*. These two spaces are then intersected, ruling out invalid modifications. The technique was evaluated on Defects4J, being able to fix 34 bugs, 13 of those the authors believe have never been fixed by other APR techniques.

**Ghambari et al.** [13] introduce an APR technique that works at Java virtual machine (JVM) bytecode level, PraPR. This simplistic method can complement other existing techniques since, for each suggested patch, there is no need for compilation to validate it. Working at such a low level allows the possibility of fixing a program without knowing its source code, not needing to learn/search for information. These approaches dramatically increase PraPR's efficiency (over 10x). The authors show that simple bytecode mutations can complement state-of-the-art techniques in *effectiveness*, *efficiency*, and *applicability*. The results of experimenting with this tool on Defects4J are shown in Figure 3.2, showed a ratio between plausible fixes and correct fixes, with a great increase in efficiency when compared to other techniques.

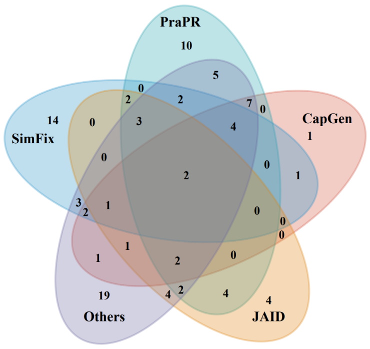

Figure 3.2: Distribution of the bugs that can be successfully fixed by PraPR and other APR techniques.

**Liu et al.** [23] introduce a template-based automatic program repair TBar. This tool integrates a total of 35 fix patterns. The authors collected these manually, identifying the patterns that other APR tools use and manually assessing them in 15 different categories, as shown in Table 3.2. The authors believe that such a catalog of fix patterns can help improve APR overall performance. However, it was noticed that fault location noise could significantly impact this tool's performance. Two different experiments were conducted on the Defects4J dataset: (1) providing the perfect bug location to TBar, avoiding the bias that fault localization can introduce, and (2) evaluating TBar in a normal program repair scenario. The results validate the authors' hypothesis showing that fix patterns are susceptible to false-positive fault locations. The authors clarify that TBar's overall performance in the benchmark experiment is hindered by a naive search strategy to select the most relevant fix ingredients to build correct patches from the patterns.

Table 3.2: *Liu et al.* [23] Analysis of Collected Patterns.

| **Fix Pattern** | **Change Action** | **Change Granularity** | **Bug Context** | **Change Spread** |
|---|---|---|---|---|
| FP1 | Insert | statement | cast expression | single |
| FP2.1 | Insert | statement | a variable or an expression returning non-primitive-type data | single |
| FP2.(2,3,4,5) | | | | dual |
| FP3 | Insert | statement | element access of array or collection variable | single |
| FP4.(1,2,3,4) | Insert | statement | any statement | single |
| FP5 | Update | expression | class instance creation expression and clone method | single |
| FP6.1 | Update | expression | conditional expression | single |
| FP6.2 | Delete | | | |
| FP6.3 | Insert | | | |
| FP7.1 | Update | expression | variable declaration expression | single |
| FP7.2 | Update | expression | cast expression | single |
| FP8.(1,2,3) | Update | expression | integral division expression | single |
| FP9.(1,2) | Update | expression | literal expression | single |
| FP10.1 | Update | expression, or statement | method invocation, class instance creation, constructor, or super constructor | single |
| FP10.2 | | | | |
| FP10.3 | Delete | | | |
| FP10.4 | Insert | | | |
| FP11.1 | Update | expression | assignment or infix-expression | single |
| FP11.2 | Update | expression | arithmetic infix-expression | single |
| FP11.3 | Update | expression | instance of expression | single |
| FP12 | Update | expression | return statement | single |
| FP13.(1,2) | Update | expression | variable expression | single |
| FP14 | Move | statement | any statement | single or multiple |
| FP15.1 | Delete | statement | any statement | single or multiple |
| FP15.2 | Delete | method | any statement | single or multiple |

**Assiri and Bieman** [6] propose a prototype tool, MUT-APR, which uses mutations to binary operators to patch programs. It is built on top of GenProg [22], using genetic programming to find a patch to a faulty program but distinguishes itself in the mutation step. The authors divided the operators into different classes, only changing the wrong operators with ones in the same category. There are four operator classes: (1) *relational*, (2) *arithmetic*, (3) *bitwise*, and (4) *shift operators*. This tool presents some limitations like the inability to fix logical operators (e.g., `||` and `&&`) and the equality operator (==). In previous work [5], the authors conducted a study to validate this technique, comparing its results against GenProg in programs from the Siemens Suite [1]. MUT-APR was able to repair faulty operators in 87.03% of the programs, while GenProg only fixed 31.48%.

**Ke et al.** [18] propose SearchRepair. The authors built this tool under the assumption that human-written code, partly satisfying a specification (test cases), "is more likely to satisfy the unwritten specification than a randomly chosen set of smaller edits generated with respect to the same partial specification." Following this idea, SearchRepair is a semantic-based code-search APR approach. It makes use of existing open-source projects to find patches for faulty programs. After localizing the possible bug, an input-output profile characterizing the expected behavior is constructed. A search for potential patches is done in the encoded database of human-written code fragments and are validated. This technique is not finished since the author's point some flaws: (1) global variables are not considered, and (2) can't handle code fragments with loops. *Ke et al.* conducted a study using the ManyBugs benchmark [21] where SearchRepair managed to fix (19%) of them, achieving a higher patch quality than other three tools.

**Kim et al.** [19] state that genetic programming based APR techniques generate invalid and senseless patches due to the randomness imposed by program mutations. To tackle this, they propose PAR, a pattern-based automatic program repair. The authors made a manual evaluation of human-written patches and categorizing them, resulting in eight common fix patterns shown in Table3.3. These patterns are used to generate code modifications until a patch for a code segment is found. To evaluate this APR's performance, six open-source projects were analyzed for replicable bugs and compare the results against GenProg [22]. PAR was able to correctly fix 27 out of 119 bugs while GenProg successfully patched 16. Also, to compare the suggested patches similarity to human-written ones, a user study of 253 participants (students and developers) was conducted, showing that PAR's resulting patches were more comparable to human patches than GenProg.

**Mechtaev et al.** [26] present Angelix. The authors' objective is to tackle semantic-based techniques' main setback, *low scalability*, introducing the concept of *angelic forest* — a semantic signature for repair — based on *angelic value* [11]. This is possible by keeping this *angelic forest*, size-independent from the problem's dimension. *Mechtaev et al.*'s notion that this *semantic signature* containing enough semantic information to make Angelix able to fix multiline bugs. The authors conducted a series of experiments to prove Angelix's scalability and patch quality when comparing against GenProg [22].

Table 3.3: PAR's fix patterns [19].

| **Fix Patterns** |
|---|
| Altering method parameters |
| Calling another method with the same parameters |
| Calling another overloaded method with one more parameter |
| Changing a branch condition |
| Adding a null checker |
| Initializing an object |
| Adding an array bound checker |
| Adding a class-cast checker |

**Nguyen et al.** [30] present SemFix, a semantic-based technique that can synthesize a patch for a faulty program even if the repair code doesn't exist in its source code. This technique retrieves a list of possible statements that are likely to contain the bug. For each of these statements, a *repair constraint* is generated with the specification for its correct behavior. Then *program synthesis* is used to try to solve this constraint. The authors compare SemFix against genetic programming-based repair techniques achieving higher *effectiveness* and *efficiency*.

**Qi et al.** [35] introduce RSRepair, an APR based on GenProg [22], but using a random search algorithm instead of genetic programming. To further improve RSRepair's efficiency, the authors optimized the number of test executions needed to find a potential fix with a *prioritization technique*. An experiment was done to compare random search algorithms' performance against genetic programming, comparing RSRepair with GenProg on faulty problems. This experiment results show that RSRepair requires fewer patch trials to find a plausible patch over genetic programming.

**Le Goues et al.** [22] present GenProg, using genetic programming to *evolve* a faulty program to achieve the desired functionality. A sequence of source code edits is used to represent a candidate patch to fix a given bug. *Crossover* and *mutation* operations are used to generate new candidate repairs until one provides the wanted functionality while fixing the program's bug. The authors built this tool under the assumption that if a problem contains a bug in a code fragment, its correct behavior is implemented elsewhere in the software source code. This aids GenProg's efficiency reducing the possible *search space*. *Le Goues et al.* implement three different mutator operators to modify statements from the program itself: (1) *deletion*, (2) *insertion*, and (3) *swap*. Experiments were done in C programs showing that GenProg can repair several errors efficiently.

**Long and Rinard** [25] present Prophet, a novel program repair system that learns a probabilistic model of correct code using machine learning techniques. This model comes from a set of successful human-written patches retrieved from open-source repositories. This model is used to rank candidate solutions based on their likeness to be correct. Prophet was evaluated on a benchmark containing 69 real-life bugs outperforming other APR tools.

**Pei et al.** [32] present AutoFix, the latest implementation of a tool AutoFix-E [41] developed in 2010. This tool generates fixes to faulty programs annotated with contracts, which can take several forms: (1) *preconditions*, (2) *postconditions,* and (3) *class invariants*. The authors use *fix actions*, to change the faulty program state. These fix actions are obtained using two different approaches: *setting*, modifying the value of variables, and *replacement*, modifying the value of expressions. AutoFix uses four templates that inject these fix actions into *suspicious* statements, as shown in Figure 3.3. *Pei et al.* conducted experiments in faulty software generating fixes for 42% of the faults. Besides, 59% of these patches are equivalent to human-written ones.

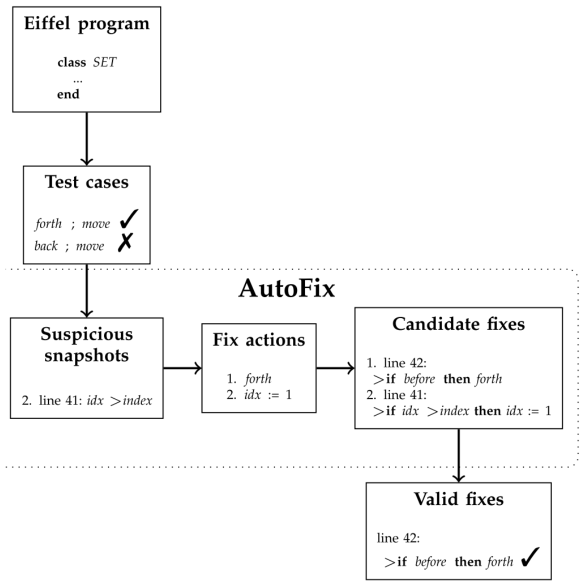

Figure 3.3: Flowchart representing the approach presented by *Pai et al.* [32].

### 3.1.3 Results Categorization

The mentioned tools, while trying to tackle the same problem, present different solutions. Therefore, they were categorized based on their approaches towards:

1. **General approach:** The are two main approaches, as explained in Section 2.1.2, that APR tools can follow. This value can either be: (1) *generate-and-validate*, or (2) *semantics-based.*

2. **Code modification:** One crucial step in any APR is how and which program elements it decides to change when repairing it. These modifications can involve code synthesis, code mutations, or AST modifications.

3. **Patch generation:** APR tools present several patch generation techniques. Possible values are *atomic change*, *pre-defined templates*, *example-based templates*, or *program synthesis*.

4. **Specification:** A specification of the program's correct behavior need to be provided to any APR tool. As explained in Section 2.1.1, this value can either be *test-specification*, or *formal-specification*.

The categorization of SLR results is described in Table 3.4.

Table 3.4: APR categorization.

| **Approach** | **Tool** | **Code modification** | **Patch generation** | **Specification** |
|---|---|---|---|---|
| generate-and-validate | ARepair [40] | expression synthesis | atomic change | test-specification |
| | CapGen [42] | AST replacement/insertion/deletion | | |
| | PraPR [13] | JVM bytecode mutation | | |
| | MUT-APR [6] | binary operator modification | | |
| | RSRepair [35] | statement mutation | | |
| | GenProg [22] | AST reuse/insertion/deletion | | |
| | sharpFix [44] | code reuse (AST replacement/insertion/deletion) | example-based templates | test-specification |
| | SimFix [15] | code reuse (AST replacement/insertion) | | |
| | PAR [19] | AST replacement/insertion/deletion | | |
| | TBar [23] | AST replacement/insertion/deletion | pre-defined templates | test-specification |
| | Prophet [25] | condition modification/synthesis | | |
| | AutoFix [32] | AST replacement | | formal-specification |
| semantics-based | SearchRepair [18] | AST synthesis | program synthesis | test-specification |
| | Angelix [26] | patch synthesis | | |
| | SemFix [30] | expression synthesis | | |

### 3.1.4 Analysis

In section 3.1.3, we can verify that the papers retrieved by the SLR can be categorized regarding the approach followed to fix a faulty program. These approaches present promising results in producing valid patches but still face some drawbacks that can influence their validity.

#### 3.1.4.1 Categorical Analysis

**General approach** Generate-and-validate is the most adopted APR approach, with only SearchRepair [18], Angelix [26], and SemFix [30] being the only 3 tools that follow a semantics-based approach.

**Code modification** When choosing which program's code elements will be a target as possible fixes to a faulty program, most tools adopt some AST variation (either by replacing, inserting, deleting, or synthesizing). Semantics-based tools prefer synthesizing a code modification, while generate-and-validate ones prefer more straightforward code modifications, like AST modifications, statement mutations [4, 31] and code reuse.

**Patch generation** Semantics-based tools generate their patches by synthesizing a new program, while generate-and-validate approaches use one of three generation techniques: (1) atomic change, (2) example-based templates, or (3) pre-defined templates, as explained in Section 2.1.2.

**Specification** When specifying a program's behavior, test suites are the most adopted choice, with only AutoFix [32] resourcing to formal-specifications.

#### 3.1.4.2 Survey Research Questions

**SRQ1:** *Which are the most relevant APR approaches?*

Most APR tools either follow a search-and-validate methodology or a semantics-based one. The main difference between them is how they approach the repair process. These approaches can range from a simple binary modification [6] to a full abstract syntax tree (AST) replacement [18].

**SRQ2:** *How is the performance of an APR approach evaluated?*

When measuring and APR performance, three factors are taken into account: (1) *scalabilty* (should scale to large projects), (2) *efficiency* (should find a correct patch in a short time), and (3) *effectiveness* (should produce repairs to different types of bugs). Analyzing these three factors provides good insight on the tools performance. It allows us to measure the quality of the suggested patches, the time required to generate such patches, and the scale of the projects that can use this tool without hindering its performance.

Achieving high performance in these three properties proves to be complicated. The way that APR techniques leverage each of these properties' importance is how they distinguish themselves from others, being more applicable in specific scenarios.

**SRQ3:** *How are APR tools validated?*

The presented approaches conduct experiments to evaluate its performance when generating patches. These are mainly tested in benchmarks, open-source projects that offer bug tracking tools, and user studies.

When using benchmarks the results obtained might not generalize to complex systems since they were only tested on a specific group of well-known bugs. When doing a user study, the level of knowledge of the participants must be taken in consideration.

**SRQ4:** *Do APR tools provide live fixes to developers?*

Only AutoFix [32] provides live bug fixing suggestions to developers since a plugin for the EiffelStudio IDE [33] was developed. With this plugin, AutoFix managed to achieve a liveness level 5 in the hierarchy proposed by Tanimoto [38]. All other tools are not able to accomplish this or don't provide any information about this characteristic.

We know of pAPRika [9, 10, 36], a Visual Studio Code extension that can provide live semantic feedback to the user. This APR tool will be analyzed in detail in the next section.

## 3.2 Summary

Section 3.1.1 describes the Systematic Literature Review methodology that was followed to gather information about the state of the art of automatic program repair tools.

The results obtained were analyzed in Section 3.1.2. Each APR implementation is summarized, describing the approaches taken during their development. They were then categorized in Section 3.1.3.

An analysis was conducted in Section 3.1.4, analysing the results obtained from the categorization and answering the previously defined Survey Research Questions.

# Chapter 4

# Problem Statement

This chapter describes the problem tackled in detail. An overview of pAPRika, a live APR tool, is presented in Section 4.1, followed by the problem under study in Section 4.2. An hypothesis to guide the development of this dissertation was build in Section 4.3, and it's Research Questions in Section 4.4. The validation methodology is presented in Section 4.5. Finally, in Section 4.6 this chapter's contents are summarized.

## 4.1 pAPRika

Campos [9, 10], noticing the lack of Automatic Program Repair tools integrated with programming environments and capable of providing live semantic suggestions to developers, as seen in Section 3.1.4, decided to tackle this problem and proposes a Visual Studio Code[1] extension. This tool merges the Automated Program Repair and Liveness research areas to provide semantic code fixes to developers that use the Visual Studio Code programming environment, achieving a level 5 in the liveness hierarchy [38], corresponding to *tactically predictive* feedback.

In a later work, Ramos [36], expanded this tool's capabilities, adapting it to follow the Language Server Protocol (LSP), a protocol that is used across IDEs, improving the patch generation technique used, and expanding it's support to other frameworks and programming languages (*JavaScript* and *TypeScript*).

[1]Visual Studio Code: https://code.visualstudio.com/

### 4.1.1 Overview

To successfully provide valid live suggestions that can fix a faulty program, pAPRika follows a generate-and-validate approach. It leverages the power of atomic changes, mutating the program's Abstract Syntax Tree to generate *plausible* patches, and the power of test-specifications, to validate them by running the available test suite, to get only the *correct* patches, which are then suggested to the developer. In Figure 4.1, pAPRika's flowchart is depicted.

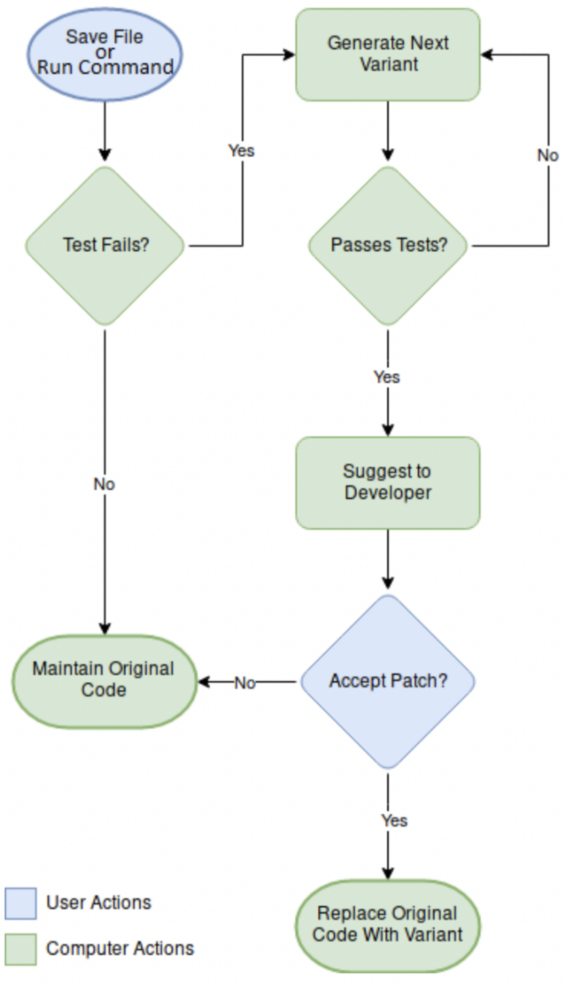

Figure 4.1: pAPRika flowchart [36].

This APR tool is a background process that automatically runs the provided test suite, trying to find a program variation that passes all the tests. Such solution is then suggested to the developer. These variations come from mutations of 7 code ingredients: (1) switch mutations, (2) parentheses mutations, (3) off-by-one mutations, (4) operator mutations, (5) boolean mutations, (6) remove prefix mutations, and (7) statement moving mutations.

### 4.1.2 Issues

This APR tool faces some problems that the authors identified:

- **Fault Localization**: The current implementation of pAPRika has no fault localization technique. The tool generates variations to the whole program every time it is run. This gives pAPRika low scalability and can hinder its liveness aspect since it should find a correct patch quickly.
- **Program Repair Strategies**: Campos [9] believes that exploring new, different, and potentially more complex strategies could increase the tool's potential regarding of how many bugs it can find in a short time.
- **Fix Ranking**: The tool is capable of generating, validating, and suggesting fixes to developers, but these patches can sometimes not be seen as *correct* to a human. Having a heuristic to rank generated patches based on *human correctness* could significantly improve its usability.
- **Visual feedback**: The tool lacks in providing helpful visual cues to the users regarding its current state.

## 4.2 Problem Definition

Software debugging is proving to become considerably more time consuming [8]. Current software development environments mainly provide syntactic suggestions to the developers when writing code, giving no relevant semantic information.

As far as our knowledge goes, almost no APR tools are integrated to programming environments and can provide live semantic code fixes to developers in an automatic way. We know about two tools that can accomplish this: (1) AutoFix [33], and (2) pAPRika, that is described in Section 4.1.

Taking a deeper look into pAPRika, it is noticeable that it suffers from some drawbacks that affect this tool's performance and usability. These problems arise from the nonexistence of a fault localization technique and its necessity for user input (either by saving a file or running a command) to start the whole process.

All things considered, having to manually run pAPRika each time we want a suggestion and running the whole test suite for obtaining it, defeats the tool's purpose of providing semantic suggestions to the users in a live and effortless way.

## 4.3 Hypothesis

Fault Localization is a whole research area in itself. Detecting a faulty code segment in the whole source code proves to be a difficult task to achieve. Since pAPRika is running simultaneously to the programmer's development we propose to build this dissertation around the following hypothesis:

> *Implementing a Fault Localization technique resorting to Reverse Code Coverage methods can improve the overall performance of live APR tools, reducing the number of tests run each time.*

To build this hypothesis, we make the assumption that, when developing a program, the programmer is most likely to insert unwanted behaviors in functions that he is working on. Making use of reverse code coverage it is possible to find which tests execute which lines of code.

This performance is not related to the quality of the suggested patches but is directly related to the time required to provide them. A lot of pAPRika's effort goes into running the test suite to detect the function that does not pass the provided specification.

A fault localization algorithm would soften this problem, since only tests for functions that the developer changed will be run, reducing the time required to generate code fixes while improving its liveness aspect.

## 4.4 Research Questions

To guide the development of this dissertation, we set the following Research Questions:

**RQ1** *Is it possible to use fault localization techniques to limit the number of tests needed to run in a test-based APR tool?* The performance of a live APR tool is directly correlated to the number of tests that are run each time. Identifying which tests are required to run can result in a great increase in performance.

**RQ2** *When limiting the number of tests on a test-based APR tool using a FL technique, are the remaining tests still enough to allow detection and fixing of bugs?* Reducing the number of tests required to run can raise the problem that not enough tests are being tested each time.

## 4.5 Validation

To validate the proposed hypothesis and answer the research questions, an empirical test was undertaken. A group of students of software engineering performed a task in Visual Studio Code using the pAPRika extension that uses fault localization.

When developing their solution, all information required to replicate programmer's approach to solving the problem was sent via telemetry to a Microsoft Azure[2] server. Alongside this information that allows the *replay* of every programmer action, information regarding pAPRika's functionality was also being sent.

With this information, we can replicate the programmers' development manually, using a modified version of pAPRika, where fault localization wasn't used. This allows for an immediate comparison between both approaches, with and without fault localization, results.

A thorough account of the validation process can be found in Chapter 6.

[2]Microsoft Azure: https://azure.microsoft.com/pt-pt/

## 4.6 Summary

Facing the lack of live APR tools detected in Chapter 3, pAPRika, an extension for Visual Studio Code development environment, was built. This tool is analyzed in Section 4.1.

Based on this tool's issues, a problem was defined in Section 4.2. The hypothesis for this work is presented in Section 4.3 stating that pAPRika's performance can be improved by implementing a fault localization technique that uses reverse code coverage methods as it's base concept. Then, in Section 4.4, we formulate Research Question that guide the development of this work.

In Section 4.5 we present our experimental methodology that will help validate our hypothesis and answer the proposed Research Questions. Our solution will be tested comparing its performance using fault localization against one without it.

# Chapter 5

# Proposed Solution

This chapter goes into detail of the proposed solution. Section 5.1 presents an overview of the solution, which is then described in the next sections. Section 5.2, details the implementation of the fault localization mechanism, as well as, each of its steps, Section 5.2.1 presents how the reverse code coverage is implemented, Section 5.2.2 explains how the detection of code changes is done, and Section 5.2.3 shows the modifications made to the test runner. In Section 5.3, changes to the extension's functionality are shown. The current know limitations are described in Section 5.4. Finally, in Chapter 5.5, this chapter is summarized.

## 5.1 Overview

In Chapter 4, we point out the problems that pAPRika[9, 36] face, regarding the lack of fault localization techniques and the fact that it is needed user input in order to run the APR tool, when offering true *semantic* suggestions to developers.

We propose that this solution is able to mitigate both of this problems, by implementing a fault localization algorithm, that uses reverse code coverage (RCC) techniques. This allows us to get the test cases that test a given function, discarding all non-relevant ones.

Assuming that a developer is more likely to introduce unwanted behaviors in functions that are being implemented, we can filter the test suite to only run tests relevant to those specific functions. By keeping track of all the updated functions in the source code, this method should increase pAPRika's overall performance providing faster code fixes to developers.

To tackle the necessity to manually run the APR process, it will be automatically run every time the development process is stopped for a few seconds, and automatically update the reverse code coverage graph that keeps track of what tests should be run per function.

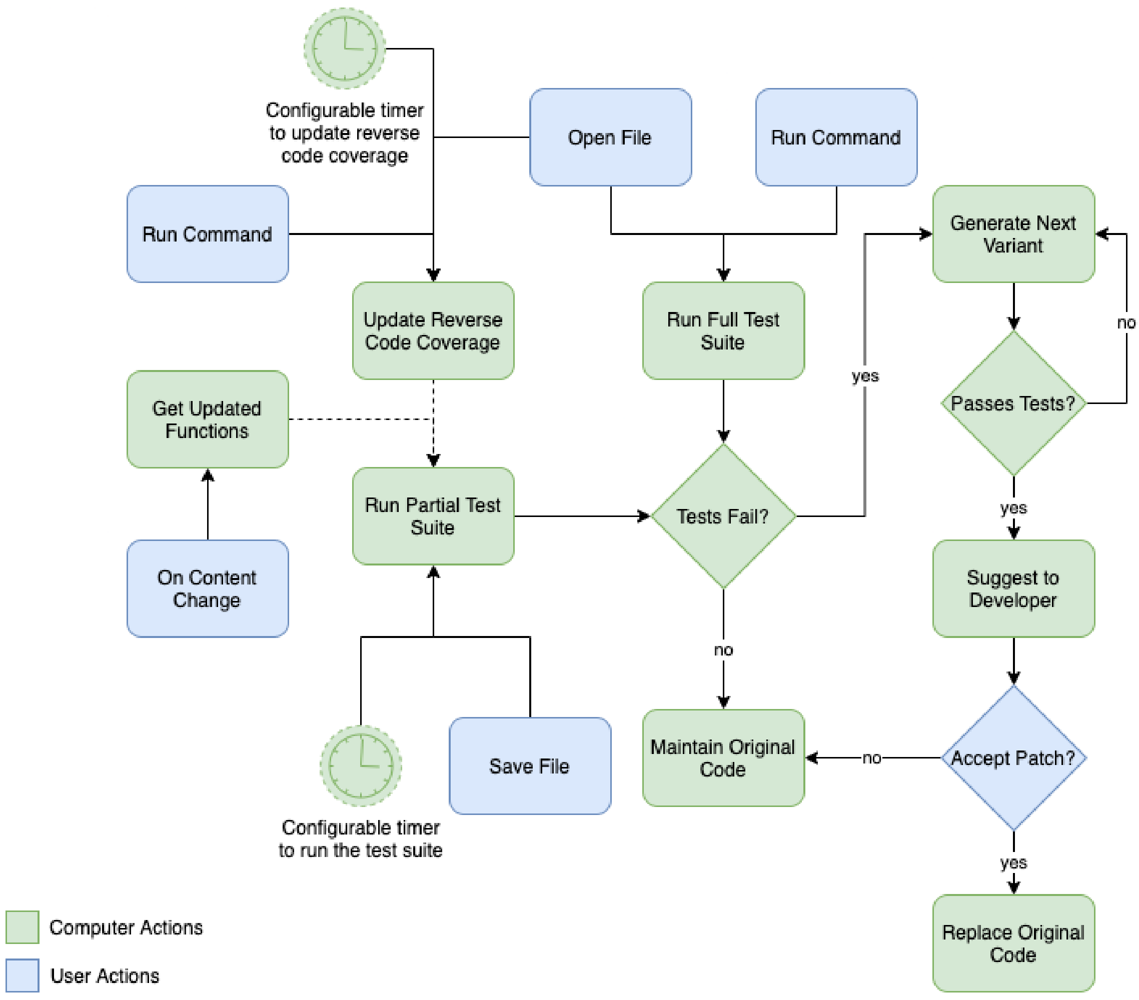

Figure 5.1: Flowchart describing the proposed solution.

In Figure 5.1, an overview of the proposed solution is depicted. At any time, when writing source code, pAPRika will automatically generate code variations for failing test and suggest them to the developer if they are successfully validated against the test suite.

An in-depth description of the implementation of the fault localization technique used to filter out unnecessary tests is given in Section 5.2, while Section 5.3 presents how we can automate the running of the extension.

## 5.2 Fault Localization

Fault localization, which focus on identifying segments of code that may contain faults, has been a laborious and time consuming task to achieve [39]. Since pAPRika is a live APR tool, we assume that the faulty code segment is contained in one of the functions changed by the developer.

Improving pAPRika's performance is a matter of reducing the number of tests from the suite necessary to run. To achieve this, we resource to reverse code coverage techniques. These techniques allow us to obtain the tests that validate a specific code segment (e.g., functions).

Knowing which functions were modified by the developer before running the automatic program repair, it is possible to get which tests are sufficient to run to find for a possible bug in the program's source code.

This implementation follows a three step approach:

- **Reverse Code Coverage**: Reverse code coverage allows us to answer the following question: *Given this segment of the source code, which tests execute it?*
- **Function Modified Detection**: When the programmer is developing a given program's functionality, it is imperative to keep track of which functions were modified. Using theses altered functions and intersecting them against the RCC previously calculated, we can determine which tests should be run.
- **Test Runner**: Having the list of tests to run, we can send them to our test runner, which is responsible to ensure that only those are run.

The following sections explain in detail how all of those three steps are implemented.

### 5.2.1 Reverse Code Coverage

Reverse code coverage allows the possibility to check how code changes affect the overall program without running the entirety of the test suite, testing only the affected spots.

The proposed approach uses the TypeScript Compiler API[1]. This API provides multiple ways to interact with the TypeScript programming language and, by extension, with JavaScript. In order to further understand the API the following concepts must also be understood:

- **Node**: Represents a node in the program's AST.
- **FunctionDeclaration**: Is a subclass of `Node` and represents the declaration of a function defined in the following format:

```
function name([param[, param[, ... param]]]) {
    statements
}
```

- **ArrowFunction**: Is a subclass of `Node` and represents the declaration of an arrow function defined in the following format:

[1]TypeScript Compiler API: https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API

```
([param[, param]]) => {
    statements
}

param => expression
```

These are nameless functions that can be assigned to variables, giving them their name.

- **MethodDeclaration**: Is a subclass of `Node` and represents the method declarations in the following format:

```
var obj = {
    foo() {},
    bar() {}
};
```

- **CallExpression**: Is a subclass of `Node` and represents a call to other function, it follows the format:

```
foo([param[, param[, ... param]]])
```

This implementation is comprised of four different stages:

- function extraction
- get first-level calls from functions
- get total calls from function
- reverse function calls

These steps are going to be explained in the following sections, presenting its algorithm as a pseudocode adaptation from the developed TypeScript code for purposes of simplicity and clarity.

#### 5.2.1.1 Function Extraction

The first stage in the reverse code coverage implementation consists of, given the complete source code's parent node, iterate through it and extract all of the nodes representing any type of the functions specified previously: `FunctionDeclaration`, `ArrowFunction`, and `MethodDeclaration`. Algorithm 1 represents this step's pseudocode.

**Algorithm 1** Function Extraction

```
1: function GetFunctionNodesFromNode(node)
2:     functionNodes ← {}
3:     functionNodes.add(GetFunctionDeclarations(node))
4:     functionNodes.add(GetArrowFunctions(node))
5:     functionNodes.add(GetMethodDeclarations(node))
6:     return functionNodes
7: end function
```

#### 5.2.1.2 First-Level Function Calls

Having extracted all the functions from the source code, it is now possible to get all the direct calls made to other functions (first-level) for each one of them. To accomplish this, `CallExpression` from the TypeScript API was used. This step's pseudocode is depicted in Algorithm 2.

**Algorithm 2** Get First Level Calls

```
1: function GetFirstLevelCalls(node)
2:     functionCalls ← {}
3:     for child of GetChildren(node) do
4:         if IsCallExpression(child) then
5:             functionCalls.add(child)
6:         end if
7:         functionCalls.add(getFirstLevelCalls(child))
8:     end for
9:     return functionCalls
10: end function
```

This is a recursive function, every node child can contain a child from itself that is a function call, so this methodology allows the verification of all descendants from the principal node.

#### 5.2.1.3 Total Function Calls

The third, and most important step, is where we get the complete call hierarchy per function. This algorithm, represented in Algorithm 3, takes as input a `Map` that contains all the first-level calls for each function. This algorithm adds the directs function calls to a `Queue` and iterates through each nodes in it until it is empty. When analyzing a node, its direct calls are also added to the queue. This allows us to get all function calls made from a function at any *depth*. A dynamic programming approach was taken to make the algorithm more efficient.

**Algorithm 3** Total Calls

```
function GetTotalCalls(firstLevelCalls)
    totalCalls ← Map()
    for functionNode, functionCalls of firstLevelCalls do
        toHandle ← Queue(functionCalls)
        calls ← {}
        while toHandle.size() > 0 do
            node ← toHandle.pop()
            if node == functionNode then                ▷ Skip if analyzing itself
                continue
            end if
            calls.add(node)
            toHandle.push(firstLevelCalls[node])
        end while
        totalCalls[functionNode] ← calls
    end for
    return totalCalls
end function
```

#### 5.2.1.4 Reverse Function Calls

Since pAPRika requires that, in every test description, the function name that to be tested must be specified, if we get all the functions that call a specific one, we know which tests to run. This is a relatively simple process since we already have all the calls that a function makes, so it is necessary to *reverse* them. This step's pseudocode is shown in Algorithm 4.

**Algorithm 4** Reverse Function Calls

```
function ReverseCalls(totalCalls)
    reverseCalls ← {}
    for function, functionCalls of totalCalls do
        for call of functionCalls do
            reverseCalls[call].add(function)
        end for
    end for
    return reverseCalls
end function
```

### 5.2.2 Function Modified Detection

Through the Visual Studio Code Extension API[2], hereinafter the Extension API, it is possible to customize and extend most of the editor's functionalities. Setting the server's document synchronization capabilities to `TextDocumentSyncKind.Incremental`, we can define a custom handler to the `workspace/onDidChangeTextDocument` that, as the name suggests, every time a change is made to a file, an event following the format shown below is sent to the server:

```
{
    uri: string          \\ uniquely identifies the document
    contentChanges : {   \\ contains all the changes made to the document
        range: {
            start: {
                line: number,
                character: number
            },
            end: {
                line: number,
                character: number
            }
        }
        rangeLength?: number
        text: string
    }[]
}
```

There was another approach that could be taken, setting the document synchronization capabilities to `TextDocumentSyncKind.Full`, but this idea was quickly discarded since, instead of sending where a change occurred, it sent the whole document's text. This method significantly hampered the detection of the modified function.

This problem now resorts to the TypeScript Compiler API to detect which functions are defined in a given range. Iterating each function node from the source file and comparing its beginning and end positions to a specific range, one can determine which `FunctionNodes` were modified.

### 5.2.3 Test Runner

To run the test suite provided as the program's specification, the *Mocha Testing Framework*[3] was used. The file containing the test suite is loaded into *Mocha*, and using the `grep` API call, it is possible to filter the tests providing a regular expression.

In order to only run a specific set of tests from the suite every time pAPRika starts, both previous steps were combined. Applying the reverse code coverage to all modified functions, it is possible to obtain a new list containing all function names that should be tested. Passing this list

[2]Visual Studio Code Extension API: https://code.visualstudio.com/api/references/vscode-api
[3]Mocha Testing Framework: https://mochajs.org/

to the test runner and using the `grep` call with a regular expression matching only the provided function names, we can ensure that only a partial suite of tests is run each time.

This approach of filtering the test suite by the function names, is only possible since pAPRika requires that every test specifies what function it validates in its description (every test should contain "`#fix {foo}`").

## 5.3 Visual Studio Code Extension

As we want to provide a more automatic experience to pAPRika, requiring less user inputs, and to integrating it with the new fault localization mechanism, some changes to how the extension work were required.

Two new settings were added:

- **pAPRika.runPAPRikaInterval**: Specifies on many seconds are required, after the last text change, to automatically run pAPRika (defaults to 4 seconds).
- **pAPRika.updateReverseCodeCoverageInterval**: Indicates how many minutes are between every automatic reverse code coverage update (defaults to 4 minutes).

Alongside these new settings, two new commands, to manually update the reverse code coverage, were also added:

- **pAPRika.updateReverseCodeCoverage**: Updates the reverse code coverage for the active document.
- **pAPRika.updateReverseCodeCoverageAll**: Updates the reverse code coverage for every open document.

With these changes, it is possible for pAPRika to automatically update its fault localization system, as well as start its program repair.

## 5.4 Known Limitations

When implementing the proposed solution, some limitations were noticed. In Section 5.2.2, there was the need to use the `workspace/onDidChangeTextDocument` notification. This proved to be a problem since it is a Visual Studio Code specific notification and not a Language Server Protocol one. Because of this, the server's internal representation of a text document was not updated, but was the only way of obtaining content changes. The server's equivalent notification (`documents/onDidChangeContent`) only sent the updated full text of the document instead. A workaround for this issue was found using `TextDocument.update(document, changes).` Every change made to the file was used to update the document's internal representation. This approach came at the cost of the tool's interoperability between development environments.

Regarding the TypeScript Compiler API, a problem also emerged. When getting the complete program's AST, its source file is loaded, loading the previously saved version of that file. This proves to be a problem when getting which functions were modified since the given ranges might not be synchronized to what is stored in the disk.

The final limitation is that for every code change the whole AST is iterated through, as explained in Section 5.2.2. This can become an expensive task to perform when dealing with bigger projects.

## 5.5 Summary

In this chapter we present and discuss the implementation choices taken during the solution's development.

In Section 5.1, a simple overview of the changes done to pAPRika are shown and analyzed, where two problems were tackled: (1) the lack of a fault localization technique, and (2) the need for some user input to start the program repairing process. The first problem solution is described in Section 5.2. The assumption that, since pAPRika is a live APR tool, the buggy code segment must be contained in one of the functions that the developer is programming. This implementation follows three steps: reverse code coverage, which is analyzed in Section 5.2.1, modified function detection, in Section 5.2.2, and, finally, test runner, described in Section 5.2.3.

The Visual Studio Code extension specific changes, are analyzed in Section 5.3. These consist in addition of two new settings, two new commands, and the amortization of the APR process. This way, no user input is required to run the APR.

Our solution still faces some limitations, mainly regarding code synchronization issues. Section 5.4 goes into detail regarding this subject.

# Chapter 6

# Empirical Study

This chapter presents a description of the empirical study of the developed tool. Section 6.1 goes into detail to all the preliminary-work required to conduct the study. Secondly, in Section 6.2, the methodology followed is presented. Then, the analysis of the obtained results and a discussion is conducted in Section 6.3, followed by a listing of the threats to the study's validity in Section 6.4. Finally, a summary of this chapter is done in Section 6.5.

## 6.1 Preliminary Work

To validate our hypothesis, defined in Section 4.3, and measure to what extent our goals, defined in Section 1.4, have been reached, we devised an empirical study on which we test our solution. To ease this evaluation, some preliminary work was conducted. This work consisted of three different tasks:

- **Telemetry**: To easily monitor the tool's performance, a telemetry system that sends data to a Microsoft Azure server was implemented into pAPRika.
- **Replay System**: Extending the already implemented telemetry, information regarding the user's actions was also sent. This change allows the reproduction of the user's actions locally.
- **Base pAPRika Comparison**: A *basic* version of pAPRika (without fault localization) must be developed in order to compare the developed solution against a baseline.

### 6.1.1 Telemetry

An important step of the whole study process, is to collect and analyze data from pAPRika. Since the the developed solution was an internal upgrade to the tool, retrieving this details is a troublesome task since there is no visual feedback provided by the extension regarding them. With this in mind, a telemetry system was implemented. This implementation uses the *vscode-extension-telemetry*[1] node module. This module provides a way for Visual Studio Code extensions to send telemetry over Application Insights[2] in Azure.

Three steps were identified as necessary in order to successfully retrieve and send data from pAPRika, via telemetry:

1. **Data Collection**: The first step to monitor this tool is identifying which data is relevant to keep track of and collect it.

2. **Send Data from LSP to Client**: Secondly, it is needed to send the collected data from the LSP's server to the extension's client. This step is required since it is not possible to send telemetry data directly from the server.

3. **Send Data from Client over Application Insights**: Finally, having all the data in the client it is needed to sent it to over Application Insights.

The second step, was made possible by implementing a `TelemetryManager` that is responsible for sending all information to the client. This *manager* sends events through a `telemetry/event` notification that is sent from the server. The client, on the other hand is subscribed to these notification by listening for them. The events sent between the LSP and the client follow the format depicted below:

```
{
    name: string                             \\ event name
    properties: {[key: string]: string}   \\ properties to send
    measures: {[key: string]: number}     \\ unsued
}
```

Using the previously mentioned package and having a telemetry event in the client from the LSP, `sendTelemetryEvent(event.name, event.properties, event.measures)` is used to send that event over Application Insights.

The data collected, which is sent in specific events, is explained in Section 6.1.1.1. Each event has specific properties related to it, but they all send the `sessionUUID` property. This value is unique to each instance of pAPRika and allows to group events by it.

[1]vscode-extension-telemetry: https://github.com/microsoft/vscode-extension-telemetry
[2]Guide to set up Application Inside: https://docs.microsoft.com/en-us/azure/application-insights/app-insights-nodejs-quick-start

#### 6.1.1.1 Telemetry Events

The communication between entities (LSP, client, and Application Insights) is made using events. Each telemetry event is described as follows and contains the depicted properties (does not show the `sessionUUID` property mentioned previously):

**CONNECTED** This event is sent every time a pAPRika instance is started.

```
CONNECTED: {
    faultLocalization: boolean    \\ is fault localization active
}
```

**FILE_OPEN** This event is sent every time a file is opened.

```
FILE_OPEN: {
    document: string,   \\ file that was open
    text: string        \\ file's full text
}
```

**FILE_CLOSE** This event is sent every time a file is closed.

```
FILE_CLOSE: {
    document: string,   \\ file that was open
    text: string        \\ file's full text
}
```

**PAPRIKA.RUN** This event is sent every time pAPRIka's program repair is run.

```
PAPRIKA.RUN: {
    document: string,               \\ file where pAPRika run
    documentText: string,           \\ file's full text
    elapsedTime: number,            \\ time taken (ms)
    fullTestSuite: boolean,         \\ did it run the full test suite
    totalTestsRun: number,          \\ number of tests run
    testsRun: string[],             \\ tests run
    passedTests: string[],          \\ tests that passed
    failedTests: string[],          \\ tests that failed
    functionsToTest?: string[],     \\ functions to test
    variationsGenerated: number,    \\ number of generated variations
    variationsTestsRun: number,     \\ tests run validating variations
    variationsPassedTests: number,  \\ tests passed validating variations
    variationsFailedTests: number,  \\ tests failed validating variations
    suggestedReplacements: {        \\ suggested replacements
        start: {line: number, char: number, offset: number},
        end: {line: number, char: number, offset: number},
        oldText: string,
        newText: string,
        code: string
    }[]
}
```

**REVERSE_CODE_COVERAGE.RUN** This event is sent every time the program's reverse code coverage is updated.

```
REVERSE_CODE_COVERAGE.RUN: {
    document: string,       \\ file that updated reverse code coverage
    documentText: string,   \\ file's full text
    elapsedTime: number     \\ time taken (ms)
}
```

**QUICK_FIX.ACCEPTED** This event is sent every time the user accepts a suggestion.

```
QUICK_FIX.ACCEPTED: {
    document: string,       \\ file where it was accepted
    code: string            \\ suggestion's code
}
```

### 6.1.2 Replay System

To be able to replicate the user's actions locally, a new telemetry event was added:

**CONTENT_CHANGE** This event is sent every 5 seconds after a modification, before running pAPRika, or before updating the reverse test coverage. Contains all code changes that occurred during that time span. These are retrieved from the `workspace/onDidChangeTextDocument` notification.

```
CONTENT_CHANGE: {
    document: string,       \\ modified file
    start: {                \\ modification start position
        line: number,
      character: number
    },
    end: {                   \\ modification end position
        line: number,
      character: number
    }
  text: string             \\ modified text
}[]
```

### 6.1.3 Base pAPRika Comparison

One of our objectives is to compare the fault localization impact on the tool's performance. With this in mind, we need to replicate the user's actions on the proposed tool, in Chapter 5, against another that only differs in the fact that it does not use fault localization. The last one will be the baseline against which we will compare the values obtained.

## 6.2 Methodology

With the goal of validating the proposed hypothesis in Section 4.3, an empirical study was conducted. The study evaluated the proposed fault localization technique's impact on a live automatic program repair tool. To achieve this, we recurred to a telemetry system, explained in Section 6.1.1. This telemetry provides enough information for us to manually *replay* a user's inputs. Using this replay system, it is possible to make a comparison between two different tools by reproducing the exact conditions from one onto the other. We will be comparing the two tools: (1) the proposed solution, with fault localization, and (2) a modified version of the developed one but without its fault localization technique, detailed in Section 6.1.3.

### 6.2.1 Plan

This section describes the guidelines that the experiment follows.

**Participants** The tests consisted of implementing two features in a JavaScript project. With this in mind, the participants must have some programming knowledge and familiarity with the JavaScript language.

**Duration** This study's duration was estimated to be around 30 minutes. A brief introduction explaining the problem and what was expected from the participants was made beforehand, taking an estimated 10 minutes.

**Environment** Due to the global pandemic situation that is being faced, the study was performed online. The participants were free to use their personal computers as long as Visual Studio Code, with the developed extension installed, was used and had the latest version of *Node.js*[3] installed.

[3]Node.js: https://nodejs.org/

**Procedure** To compare the performance difference of a live APR tool using fault localization with the performance of one without it, the study was divided into two stages, the first one is performed by the participants while the second is performed by us:

- **First Part**: Every participant, before starting the study, sends us their `sessionUUID` value that would be printed in the console. Following this, they start implementing the tasks that they were asked to. All the necessary information is being sent by telemetry to a server. This part was executed using pAPRika **with** fault localization.
- **Second Part**: When the participants finish their assigned tasks, we, using their `sessionUUIDs` retrieved all their logs from the server using the following query:

  ```
  customEvents
  | where customDimensions.sessionUUID == "<sessionUUID>"
  ```

  With this information, it was possible to replicate their development process in a programming ambient **without** fault localization. All relevant information from this step is also sent through telemetry.

**Data** During this study, several data was collected to be analyzed:

- **Tests run**: The tests that were executed during the program repair.
- **Tests failed**: The tests that failed when the test suite run.
- **Tests passed**: The tests that passed when the test suite run.
- **Changed functions**: We kept track of which functions were modified by the participants.
- **Proposed suggestions**: The suggestions that the APR proposed.
- **Elapsed time**: The execution time of the APR process was being recorded but then proved to be **not useful** since it was not possible for us to accurately compare this value against our controlled environment due to computer differences.
- **Document's source code**: The document's full source code was being collected.

### 6.2.2 Tasks

This study aims not to evaluate the quality of the suggested code fixes but the improvement that the fault localization algorithm has in a live APR. So, simple tasks were proposed to the participants.

Both tasks are adaptations of a *Raspberry Pi Pico Emulator*, developed by *wokwi*, called rp2040js[4]. This project is developed in TypeScript and provides a test suite validating its instruction implementations, perfectly fitting our needs.

[4]rp2040js, by wokwi: https://github.com/wokwi/rp2040js

Each task provides the basic setup required to simulate a *Raspberry Pi Pico* as well as the implementation of some of its instructions. The goal of the tasks was to implement the `LDR (immediate)` and the `PUSH` instructions, respectively.

The specification for all — implemented and to be implemented — instructions was provided. As well as a brief overview of the program's already implemented functionalities.

## 6.3 Results

During the study, each participant was required to solve two different tasks, as described in Section 6.2.2. Posteriorly, we took replay information to solve the tasks the exact same way as the participants but using an extension version without fault localization. The following results were compared:

- Tests that run every time the program repair started.
- Functions that were changed before running the APR.

Below follows the comparison and analysis of these metrics:

### 6.3.1 Tests Run

In Table 6.1, we compare the mean number of tests run for both of the approaches. The values obtained from the tool without fault localization are expected since every time pAPRika run, the whole test suite (containing 12 tests) is also run.

These initial observations are in conformity to what was expected. Having the participants to undertake a study where the need to implement two problem (each one with one test validation them), the mean value for the total of tests run is presumed to be between one and two. This is due to the fact that the participants can either, simultaneously, tackled both problems, running the two corresponding tests, or focus on each problem at a time, running only one of the tests in the whole test suite.

### 6.3.2 Functions Changed Before Running the APR.

The detection of function changes directly affects which tests should be run. So, analyzing the performance of these detections is sufficient to evaluate the if tests run by the tool are the correct ones.

In order to analyze these results two metrics were used:

- **Precision**: Precision is a metric that quantifies the number of correct positive predictions made. Simply put, precision corresponds to the ratio of correct positive predictions out of all positive predictions made. In our case, this metric represents the number, from all the tests run, that was necessary to run.

Table 6.1: Results regarding the number of tests run.

| | With Fault Localization | | Without Fault Localization | |
|---|---|---|---|---|
| | Mean | Std. Deviation | Mean | Std. Deviation |
| #1 | 1.909 | 3.204 | 12 | 0 |
| #2 | 1.480 | 2.156 | 12 | 0 |
| #3 | 1.333 | 1.502 | 12 | 0 |
| #4 | 2.000 | 3.055 | 12 | 0 |
| #5 | 2.077 | 2.894 | 12 | 0 |
| #6 | 1.750 | 2.657 | 12 | 0 |
| #7 | 2.000 | 3.028 | 12 | 0 |
| #8 | 2.000 | 2.909 | 12 | 0 |
| #9 | 1.309 | 1.536 | 12 | 0 |
| #10 | 1.032 | 2.071 | 12 | 0 |

- **Recall**: Recall is a metric that quantifies the number of correct positive predictions made out of all positive predictions that could have been made. Here, this represents the number, from all the tests that should run, actually run.

The results presented in Table 6.2, depict the precision and recall regarding the detection of code changes. The precision and recall values retrieved are pretty good, getting a mean of $0.95865 \pm 0.0186$ and $0.9871 \pm 0.0142$, respectively.

### 6.3.3 Discussion

This experiment was taken in order to validate the following hypothesis:

> *Implementing a Fault Localization technique resorting to Reverse Code Coverage methods can improve the overall performance of live APR tools, reducing the number of tests run each time.*

With it we pretend to answer the Research Questions presented in Section 4.4:

**RQ1:** *Is it possible to use fault localization techniques to limit the number of tests needed to run in a test-based APR tool?*

In order to answer this question, an evaluation comparing the number of tests run with and without fault localization was done.

Using the replay system, described in Section 6.1.2, we are able to compare the test execution process in the exact same state between the two tools. The answer to this Research Question lies in Section 6.3.1. In the mentioned section, we can see that the whole test suite is comprised of 12 individual tests, which are always run in the implementation not using fault localization.

Table 6.2: Results regarding the function change detection.

| | Function Changes | | | | |
|---|---|---|---|---|---|
| | True Positives | False Positives | False Negatives | Precision | Recall |
| #1 | 24 | 0 | 0 | 1.0000 | 1.0000 |
| #2 | 48 | 2 | 2 | 0.9600 | 0.9600 |
| #3 | 91 | 4 | 2 | 0.9579 | 0.9785 |
| #4 | 32 | 1 | 0 | 0.9697 | 1.0000 |
| #5 | 34 | 2 | 1 | 0.9444 | 0.9714 |
| #6 | 41 | 1 | 1 | 0.9762 | 0.9762 |
| #7 | 52 | 3 | 0 | 0.9455 | 1.0000 |
| #8 | 64 | 4 | 1 | 0.9412 | 0.9846 |
| #9 | 84 | 6 | 0 | 0.9333 | 1.0000 |
| #10 | 46 | 2 | 0 | 0.9583 | 1.0000 |

In the proposed solution, the number of tests run, was significantly lower, and between the suspected interval (between one and two tests). This happens since the study was composed of two different tasks, each task with one test validating it.

So, we can argue that there is enough evidence to support the idea that, using fault localization we can reduce the number of tests run in a live APR tool.

**RQ2:** *When limiting the number of tests on a test-based APR tool using a FL technique, are the remaining tests still enough to allow detection and fixing of bugs?*

The answer to this question, RQ2, is more complex than the previous one. We needed to resource to the replay system, and manually, simulate every code change made by the participants in a controlled environment.

Every code change position was analyzed, and the functions in that location were tracked as *modified functions* until the program repair process began. These functions were then compared against the functions in the field `functionsToTest` that was sent as part of the `PAPRIKA.RUN` telemetry event. Comparing these values gives us a confusion matrix, and we can calculate the *precision* and *recall* metrics. These metrics provide information relative to the number of modified functions correctly predicted. These metrics are explained in detail in Section 6.3.1.

## 6.4 Threats to Validity

As with any empirical study,certain threats to its validity must be considered and the respective impact in the study [34].

In this section three validity threats are analyzed: (1) *Construct Validity*, (2) *Internal Validity*, and (3) *External Validity*. The following sections describe each of these threats.

### 6.4.1 Construct Validity

*Construct Validity* analyzes to which degree the inferences made can validly be made in our study. This type of validity is related to generalization.

**Task's Validity**

This is, most likely, one the most relevant threats to construct validity in this study. The fidelity of which the tasks used in this experiment represent represent real-life scenarios may impose a threat to the validation of this experiment. The tasks used were adapted from an open source repository, tackling a very specific area, instead of specially made for this purpose.

**Hypothesis Guessing**

Being under a test environment might make some participates try to guess what we are trying to demonstrate and change their behavior based on that assumption. So, the variables retrieved during the study might not be as accurate. While we cannot be sure that a participant is trying to obtain a specific result, we decided not to divulge the hypothesis and objectives of the study.

### 6.4.2 Internal Validity

*Internal validity* reflects how a given study ensures the elimination of alternative explanations for a finding.

**Telemetry Dependability**

Since we make heavy usage of the telemetry system to aid the development of this study, we have become heavily dependent on it. Therefore, losing this system, even for a small amount of time, could result in the loss of events. With this in mind, the participants tried to perform the study in locations with a good internet connection. Microsoft's server was checked before each participant evaluation to see its status.

**Manual Work**

Considering the intense manual labour, one can argue that the obtained results may contain some error in them. We accept that this can be a possibility, but the necessary work was carefully done.

### 6.4.3 External Validity

The final type of validation discussed is *External Validity*. This refers to how well the the results obtained from the study can be generalized to other settings.

**Sample Size**

In this study we apply the process described in Section 6.2.1 to two different tasks among 10 participants. We are aware that the size of tests conducted is low in order to confidently make assumptions regarding the improvement made by using a fault localization technique. Notwithstanding, the findings exhibited by the study reveal the potential in using a fault localization technique based in reverse code coverage approaches in live automatic program repair tools.

**Practical Situations**

This study conducted experiments on relatively small sized projects. Therefore, using the developed solution in a bigger project (with more tests and functions) may not achieve the same results as the ones that we obtained during this validation. Regardless, future experiments should make use of bigger projects.

## 6.5 Summary

This chapter starts by presenting the preliminary work necessary to implement, in Section 6.1, to conduct the empirical study successfully. This work is composed of three different tasks, detailed in its subsections.

An explanation of the experimental study is presented in Section 6.2. Here, the methodology followed, overall plan and tasks are described in detail. The results are delineated in Section 6.3, accompanied by a discussion and interpretation of the obtained results regarding the proposed hypothesis and research questions.

Threats to this study's validity are presented in Section 6.4, showing the approaches taken to reduce their impact.

# Chapter 7

# Conclusions

This chapter presents the considerations on the work developed during this document. Section 7.1 presents the main conclusions of this work, revisiting the hypothesis and research questions. Secondly, Section 7.2 details the main contributions made with our work and, finally, in Section 7.3 an analysis of work to be tackle in the future is done.

## 7.1 Conclusions

Software development is becoming increasingly complex. Following this trend, software debugging is also increasing in difficulty [8]. Since the latest one is essential to the first one, two research areas have emerged to ease this stage of software development: (1) Automatic Program Repair and (2) Live Programming. The first one's base idea is to try to fix a faulty program by generating code fixes. The second one was built around automatically providing live feedback to programmers about what they are developing. However, the systematic literature review showed that very few tools managed to merge these two areas into one tool, achieving live *semantic* feedback. Moreover, the tools that managed to accomplish this are still in early development and lack some crucial features to ensure efficiency and usability, namely a fault localization technique.

In our work, we address this problem by implementing a fault coverage approach based on reverse code coverage techniques, improving the tool's overall performance, as initially set by our hypothesis:

> *Implementing a Fault Localization technique resorting to Reverse Code Coverage methods can improve the overall performance of live APR tools, reducing the number of tests run each time.*

We implemented the proposed fault localization technique in a live APR tool [9, 36] integrated into the Visual Studio Code developing environment to validate our hypothesis. Then, an empirical study was performed to evaluate the performance of the proposed solution.

This experiment was used to validate the mentioned hypothesis and answer the following Research Questions:

**RQ1:** *Is it possible to use fault localization techniques to limit the number of tests needed to run in a test-based APR tool?*

We argue that a fault localization implementation improves the APR performance by reducing the number of tests required. This is supported by a direct comparison between two tools, in the same conditions, only differing because one uses the fault localization techniques and the other doesn't. Comparing the mean number of tests run every time the program repair process started between both tools, we can see a significant decrease in tests run when using fault localization.

**RQ2:** *When limiting the number of tests on a test-based APR tool using a FL technique, are the remaining tests still enough to allow detection and fixing of bugs?*

Our implementation is built around the assumption that a programmer is more likely to introduce a bug in functions that he is developing. With this in mind, knowing the modified functions and resourcing to reverse code coverage, we can get which tests should be run. Using the replay system, we can reproduce a participant's inputs. This way, we can manually verify the modified functions and the ones that were tested. We obtained a precision of $0.95865 \pm 0.0186$ and a recall of $0.9871 \pm 0.0142$. These results support the fact that the tests run by the fault localization technique are sufficient.

Ultimately, we can validate our hypothesis, as we find that the fault localization technique allows a live Automatic Program Repair tool only to run a subset of the whole test suite.

## 7.2 Main Contributions

Our work resulted in the following main contributions:

**Systematic Literature Review on Automatic Program Repair:** The current state of the art of Automatic Program Repair was analyzed.

**Fault localization technique using reverse code coverage:** We present a fault localization implementation using reverse code coverage techniques. This technique was implemented in a live APR tool (pAPRika).

**Empirical study:** An empirical study with 10 participants was conducted to validate the developed fault localization technique.

## 7.3 Future Work

Our proposed solution currently contains certain limitations described in Section 5.4 that could be improved. Alongside this, different evaluations could be conducted to validate the fault localization technique further. In this section, we summarize our thoughts and suggestions for future development in this area:

- The current implementation of the fault localization technique suffers a limitation from the TypeScript Compiler API. It is only possible to extract an AST from a *saved* file. This introduces file synchronization issues since now the APR runs automatically, independently if the open file is saved or not. Fixing this could involve creating a temporary file synchronized with the server's internal representation of itself.

- When detecting which functions were modified by a code change, we iterate through the whole program's AST, significantly affecting its performance. The challenge here consists in improving this step, reducing the number of function nodes needed.

- Conducting other experiments to compare the suggested patches using fault localization against the suggestions given without it. This would allow verifying that the tool's behavior was not altered due to the fault localization technique.

# References

[1] Software-artifact infrastructure repository: Object downloads. Available at `https://sir.csc.ncsu.edu/php/previewfiles.php`. Accessed in January 2021.

[2] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 89–98, 2007.

[3] Ademar Aguiar, André Restivo, Filipe Correia, Hugo Ferreira, and João Dias. Live software development: tightening the feedback loops. pages 1–6, 04 2019.

[4] Sérgio Almeida, Ana CR Paiva, and André Restivo. Mutation-based web test case generation. In *International Conference on the Quality of Information and Communications Technology*, pages 339–346. Springer, 2019.

[5] F. Y. Assiri and J. M. Bieman. An assessment of the quality of automated program operator repair. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 273–282, 2014.

[6] Fatmah Y. Assiri and James M. Bieman. Mut-apr: Mutation-based automated program repair research tool. In Kohei Arai, Supriya Kapoor, and Rahul Bhatia, editors, *Advances in Information and Communication Networks*, pages 256–270, Cham, 2019. Springer International Publishing.

[7] R. Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, 1985.

[8] Tom Britton, Lisa Jeng, Graham Carver, Tomer Katzenellenbogen, and Paul Cheak. Reversible debugging software "quantify the time and cost saved using reversible debuggers". 11 2020.

[9] Diogo Campos. Tests as specifications towards better code completion. 2019.

[10] Diogo Campos, André Restivo, Hugo Sereno Ferreira, and Afonso Ramos. Automatic program repair as semantic suggestions: An empirical study. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 217–228, 2021.

[11] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 121–130, 2011.

[12] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, 2019.

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 19–30, New York, NY, USA, 2019. Association for Computing Machinery.

[14] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, page 662–664, USA, 2009. IEEE Computer Society.

[15] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 298–309, New York, NY, USA, 2018. Association for Computing Machinery.

[16] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, page 273–282, New York, NY, USA, 2005. Association for Computing Machinery.

[17] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 437–440, New York, NY, USA, 2014. Association for Computing Machinery.

[18] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 295–306, 2015.

[19] 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.

[20] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8each. In 2012 34th International Conference on Software Engineering (ICSE), pages 3--13, 2012.$

[21] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[22] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[23] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. TBar: Revisiting Template-Based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42, New York, NY, USA, 2019. Association for Computing Machinery.

[24] 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, 2016. Association for Computing Machinery.

[25] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *SIGPLAN Not.*, 51(1):298–312, January 2016.

[26] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.

[27] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1), January 2018.

[28] Monika A. F. Müllerburg. The role of debugging within software engineering environments. *SIGPLAN Not.*, 18(8):81–90, March 1983.

[29] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. The power of "why" and "why not": Enriching scenario exploration with provenance. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 106–116, New York, NY, USA, 2017. Association for Computing Machinery.

[30] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.

[31] Ana CR Paiva, André Restivo, and Sérgio Almeida. Test case generation based on mutations over user execution traces. *Software Quality Journal*, 28(3):1173–1186, 2020.

[32] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[33] Yu Pei, Carlo Furia, Martin Nordio, and Bertrand Meyer. Automated program repair in an integrated development environment. pages 681–684, 05 2015.

[34] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 345–355, New York, NY, USA, 2000. Association for Computing Machinery.

[35] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 254–265, New York, NY, USA, 2014. Association for Computing Machinery.

[36] Afonso Jorge Moreira Maia Ramos. Property tests as specifications towards better code completion. 2020.

[37] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object-oriented program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 648–659, 2017.

[38] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34, 2013.

[39] Iris Vessey. Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[40] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. Automated model repair for alloy. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 577–588, New York, NY, USA, 2018. Association for Computing Machinery.

[41] Yi Wei, Yu Pei, Carlo A. Furia, Lucas S. Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 61–72, New York, NY, USA, 2010. Association for Computing Machinery.

[42] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1–11, New York, NY, USA, 2018. Association for Computing Machinery.

[43] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[44] Qi Xin and Steven P Reiss. Better Code Search and Reuse for Better Program Repair. In *Proceedings of the 6th International Workshop on Genetic Improvement*, GI '19, pages 10–17. IEEE Press, 2019.

[45] Qi Xin and Steven P Reiss. Better Code Search and Reuse for Better Program Repair. In *Proceedings of the 6th International Workshop on Genetic Improvement*, GI '19, pages 10–17. IEEE Press, 2019.

[46] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, May 2018.