FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Tests as Specifications towards better Code Completion

Diogo Campos

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Restivo

Second Supervisor: Hugo Sereno Ferreira

July 25, 2019

Tests as Specifications towards better Code Completion

Diogo Campos

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Ademar Aguiar

External Examiner: Prof. Pedro Ribeiro

Supervisor: Prof. André Restivo

July 25, 2019

Abstract

Software development is inherently complex, both from a technical and a process standpoint. Over the years, progress has been made to simplify and accelerate it, with the advent of higher level programming languages and build systems on the technical side and software development processes focused on agility on the process side.

Live Programming entails the notion of providing live feedback to developers, reducing the feedback loop of the edit-compile-run software development cycle and fastening the software development process. Existing code completion tools, commonly used by developers, provide them with *syntactic* suggestions as live feedback, but still lack the ability to provide *semantic* ones.

Automated Program Repair is an area of research focused on automating the act of fixing bugs, one of the complex tasks in software development. While current solutions display promising results, they also present some limitations, namely in terms of maintainability of the code or in producing overfitted patches.

Combining both these areas, we propose a live Automated Program Repair tool, here implemented as an extension to *Visual Studio Code*, which leverages unit tests as specifications and generates code variations to repair bugs in *JavaScript* code. With this, we provide real-time *semantic* suggestions to developers while they write code, achieving a level 5 in the liveness hierarchy, corresponding to *tactically predictive* feedback. Furthermore, by doing so, we believe that such an approach would allow developers to discard those suggestions they deem as overfitted or as reducing maintainability.

The patches (candidate repairs to the faulty program) are generated by using a mutation-based Automated Program Repair technique, which enables us to generate candidate solutions in a timely manner and find fixes within the time constraints required to offer liveness as an attribute.

An empirical study with 16 participants was conducted, with results showing that a live Automated Program Repair tool improves the speed of developers in repairing faulty programs. In addition, we establish that there is a difference in the final code of developers using the extension when compared with those not using it.

The contributions of this work include the implemented *Visual Studio Code* extension as well as the aforementioned empirical study.

Resumo

O desenvolvimento de software é inerentemente complexo, tanto de uma perspetiva técnica como de processo. Ao longo dos anos, tem sido feito progresso para o simplificar e acelerar, com o surgimento de linguages de programação de mais alto nível e ferramentas de desenvolvimento do lado técnico e processos de desenvolvimento de software focados na agilidade do lado de processo.

Live Programming consiste na noção de proporcionar *feedback* em tempo real aos programadores, reduzindo o ciclo de desenvolvimento de *software* tradicionalmente composto pelas fases de editar-compilar-correr e acelerando o processo de desenvolvimento de *software*. As ferramentas de *code completion* atuais, usadas por engenheiros de software de forma regular, fornecem sugestões *sintáticas* como *feedback* em tempo real, mas não têm ainda a capacidade para fornecer sugestões *semânticas*.

Automated Program Repair é uma área de pesquisa focada em automatizar o ato de corrigir bugs, uma das tarefas complexas no desenvolvimento de software. Embora as soluções atuais exibam resultados promissores, apresentam ainda algumas limitações, nomeadamente em termos da redução da *maintainability* do código ou na produção de correções que sofrem de *overfitting*.

Combinando ambas áreas, propomos uma ferramenta *live* de *Automated Program Repair*, aqui implementada como uma extensão para *Visual Studio Code*, que utiliza testes unitários para gerar variações no código e reparar falhas em *JavaScript*. Assim, fornecemos sugestões *semânticas* em tempo real aos programadores enquanto escrevem código, atingindo um nível 5 na hierarquia de *liveness*, correspondendo a *tactically predictive feedback*. Além disso, ao fazê-lo, acreditamos que esta abordagem pode permitir aos programadores descartar as sugestões que considerem *overfitte* ou que reduzam a *maintainability*.

Os patches (candidatos a reparação do programa com falhas) são gerados usando uma técnica de *Automated Program Repair* baseada em mutações, que permite a geração de possíveis soluções de forma rápida o suficiente para permitir propô-las como *feedback* em tempo real.

Realizou-se um estudo empírico com 16 participantes e os resultados indicam que uma ferramenta *live* de *Automated Program Repair* melhora a velocidade dos programadores na reparação de programas defeituosos. Além disso, é demonstrada a existência de uma diferença no código final de programadores que usam a extensão em comparação com aqueles que não a usam.

As contribuições deste trabalho incluem a extensão implementada *Visual Studio Code* e o estudo empírico supracitado.

Acknowledgements

My sincere gratitude to my supervisors, André Restivo and Hugo Sereno Ferreira, for their guidance, knowledge, patience, and unwavering support and even more so for their continued enthusiasm and never-ending ideas.

Many thanks to Filipe Correia and João Pedro Dias, who, in spite of not being my supervisors, were always around to answer any question, help solve any problem or discuss new ideas, as well as to Sara Fernandes and Mário Fernandes, who heard my weekly rants and with whom I had the most productive discussions while drinking vending machine coffee of questionable quality.

To Carolina, who almost single-handedly kept me sane, whose love, support, patience and empathy are endless, who put up with all-nighters and mood swings, my profound gratitude.

My heartfelt thanks to my family, for none of this would be possible without them. No words can describe the support they have given me. It is a debt I can never repay.

I am forever grateful for two very special groups of friends, **random.org # Long Live Tugalândia E Os Preços Baixos* and *Os Bebados. A Ressacar Desde 96*, who were always there to give me the push I needed, even when I didn't know I needed it.

To tea, I deeply appreciate your existence.

My thanks to Filip Filipov, Iliya Trendafilov, Alex Astafiev, Bozhidar Penchev, Atanas Radkov, Albert Ferràs, Jacek Wojdel, Pedro Portela, among many others whose advice, expertise and wiseness impacted me as a professional in the five years culminating with this work.

Lastly, and because this section would neither be complete nor truthful without it, a token of appreciation to Bill Belichik and Tom Brady, who, unbeknownst to them, have been a constant source of motivation and desire to improve. Job done.

Diogo Campos

"I love deadlines. I love the whooshing noise they make as they go by."

Douglas Adams

Contents

1 Introduction **1**
1.1 Context . . . 1
1.2 Problem Definition and Goals . . . 2
1.3 Motivation . . . 2
1.4 Structure . . . 3

2 State of the Art **5**
2.1 Background . . . 5
2.1.1 Automated Program Repair . . . 5
2.1.2 Live Programming . . . 6
2.2 Automated Program Repair . . . 7
2.2.1 Formal Specifications . . . 7
2.2.2 Tests as Specifications . . . 10
2.2.3 Current Challenges . . . 14

3 Problem Statement **17**
3.1 Context and Problem Definition . . . 17
3.1.1 Automated Program Repair . . . 17
3.1.2 Editor Services . . . 18
3.1.3 Problem Definition . . . 18
3.2 Research Questions . . . 19
3.3 Validation . . . 19
3.4 Conclusions . . . 20

4 Proposed Solution **21**
4.1 Overview . . . 21
4.2 Automated Program Repair . . . 23
4.2.1 Mutant Generation . . . 23
4.3 Visual Studio Code Extension . . . 27
4.3.1 Accept User Input . . . 28
4.3.2 Display Suggestion . . . 28
4.3.3 Apply Changes . . . 30
4.4 Summary . . . 30

5 Empirical Evaluation **31**
5.1 Methodology . . . 31
5.1.1 Plan . . . 32
5.1.2 Tasks . . . 33

5.2 Results 35
5.2.1 Background 36
5.2.2 Problem Sets 39
5.2.3 Post-Test Survey 44
5.2.4 Discussion 45
5.3 Threats to Validity 47
5.3.1 Construct Validity 47
5.3.2 Internal Validity 47
5.3.3 External Validity 48
5.4 Summary 49

6 Conclusions **51**
6.1 Conclusions 51
6.2 Main Contributions 52
6.3 Future Work 53

References **55**

List of Figures

2.1 Liveness hierarchy 7
2.2 Flowchart describing the approach presented by He and Gupta. 8

4.1 Flowchart describing the proposed solution. 22
4.2 Underlined bug fix location. 28
4.3 Example of a patch suggestion in a dialog box. 29
4.4 Example of a patch suggestion in the problems tab. 29
4.5 Example of the *light bulb* with code actions. 30

5.1 Histogram of background scores of both participant groups. 37
5.2 Stacked Bar Chart with frequencies of each *Background* survey item. 38
5.3 Stacked Bar Chart with frequencies of each *Problem Set X* survey item. 42
5.4 Stacked Bar Chart with frequencies of each *Problem Set Y* survey item. 43
5.5 Stacked Bar Chart with frequencies of each *Post-Test* survey item. 45

LIST OF FIGURES

List of Tables

2.1 Comparison between the works presented by Könighofer and Bloem and Rothenberg and Grumberg 10
2.2 Comparison between number of patches generated by GenProg and MUT-APR. . 13
2.3 Comparison between patches generated by GenProg, SPR and Prophet. 14

5.1 Statistical measures and *p-values* for hypothesis tests on background scores. . . . 37
5.2 Statistical measures and MWU *p-value* of time to reach solution for each problem. 40
5.3 Percentage of participants that accepted a fix suggestion per problem. 43

LIST OF TABLES

Abbreviations

APR	Automated Program Repair
AP	Automatic Programming
AST	Abstract Syntax Tree
DFS	Depth-First Search
GI	Genetic Improvement
GP	Genetic Programming
IDE	Integrated Development Environment
LHS	Left-Hand Side
RHS	Right-Hand Side
SAT	Boolean satisfiability problem
SBSE	Search-based software engineering
SMT	Satisfiability Modulo Theories

Chapter 1

Introduction

1.1 Context . . . 1
1.2 Problem Definition and Goals . . . 2
1.3 Motivation . . . 2
1.4 Structure . . . 3

This chapter describes the context, motivation and scope of this work. In Section 1.1, an overview of the context surrounding this work is given. Section 1.2 defines the problem statement and goals, discussing what it intends to achieve and how, while Section 1.3 elaborates on the motivation behind it. In Section 1.4, the structure of the remainder of this report is laid out with a brief overview of what each section consists of.

1.1 Context

Software is everywhere and increasingly complex. In order to accommodate that ever-growing complexity, there have been advances in the software development process [FDN14], with the advent of frameworks aiming to reduce the software development cycle and tooling focused on automating steps of that process, such as build automation tools and intelligent code completion.

One field of research in particular, Automatic Programming has focused on automating the actual practice of programming. This spans a variety of techniques, all of them concerning the transition from an higher-level specification to a lower level one, that go beyond the, at the time, state of the art of compilers [RW88, Bal85].

A major component of the software development process is guaranteeing software quality with respect to failures and defects, and this task is in itself a difficult one [RT96], consuming valuable development time. Automated Program Repair aims to automatically identify, locate and fix bugs [LGDVFW12], tasks which constitute a significant part of software maintenance.

In addition to those efforts, other research areas attempt to provide frameworks allowing developers to build software faster. Live Programming, in specific, entails the notion of providing live feedback to developers. This characteristic, with the different possible levels of liveness, shortens the feedback loop of the edit-compile-run software development cycle in varying degrees [Tan13].

1.2 Problem Definition and Goals

By automating bug fixing, the time needed to execute this task can be reduced, driving down the cost of building and maintaining software. There are, however, a number of problems with current Automated Program Repair approaches preventing them from being used widely. Among them, the need for correctness and completeness of the specifications to provide the program repair techniques and the low quality in terms of readability and maintainability of machine-generated repairs.

In parallel, code completion tools have become common throughout the industry and academia alike, providing *tactically predictive* feedback to developers, which corresponds to a level 5 in *Tanimoto*'s proposed liveness hierarchy [Tan13]. One major limitation of tools of this type is the inability to offer true *semantic* suggestions [LBF+12].

Looking at the problems in both of these areas at the same time suggests they might complement each other. In fact, the lack of *semantic* suggestions can be rectified by leveraging Automated Program Repair approaches to generate them. In addition, doing so introduces a step in the Automated Program Repair process in which a human developer has to look at a generated patch and make a decision on whether to accept it or not. This way, patches with critical readability and maintainability issues could be immediately discarded.

It is this combined solution, tackling the problems of two different areas, that we aim to present in this report, alongside an empirical evaluation.

1.3 Motivation

Maintaining software is costly and often not the most attractive part of Software Engineering, but it's also of the utmost importance. Tasks like those, indispensable but presenting high costs, are perfect candidates for automation. On this basis, the outlook of Automated Program Repair is promising, with the potential to drastically reduce the cost of building software. Wide acceptance, however, is largely dependent on the existence of tools well integrated with the environments software developers are used to.

Doing so as a form of code completion makes use of an existing mechanism, and it's simultaneously a good fit in terms of incorporation of Automated Program Repair in development environments and as an avenue towards extending code completion tools to provide *semantic* suggestions.

1.4 Structure

There are 5 other chapters in addition to this introduction. In chapter 2, the state of the art regarding Automated Program Repair and liveness of existing approaches is discussed. Chapter 3 defines the problem statement, research questions and proposed hypothesis. Chapter 4 describes the proposed solution and respective implementation in detail. Chapter 5 presents a thorough account of the undertaken validation process. Chapter 6 concludes this document, summarising its conclusions and main contributions as well as a brief section on future work.

Chapter 2

State of the Art

2.1	Background	5
2.2	Automated Program Repair	7

This chapter describes the state of the art regarding the context of the work presented in this dissertation. Section 2.1 briefly explains the background concepts needed for comprehension of the later sections. In Section 2.2 a survey of current state of the art solutions in the Automated Program Repair space is given, while in Section 2.2.3 the main challenges in the area are identified.

2.1 Background

Before delving into the state of the art, a number of concepts are required for full understanding of this dissertation. Subsection 2.1.1 includes definitions of concepts related to Automated Program Repair, including those pertaining to the wider areas of Software Reliability and Automatic Programming. In Subsection 2.1.2, it is described both the liveness characteristic of software and the research area studying it.

2.1.1 Automated Program Repair

Automated Program Repair as a research area is closely related to Automatic Programming and the study of what constitutes software bugs and the process of correcting them. In the APR area specifically, different authors use different words to describe the same concepts, with this subsection serving the purpose of a summary of the terminology adopted in this dissertation.

- **Automatic Programming** is generally considered to be the transition from a higher level specification to a lower level one, although, as Balzer said, *at any point in time, the term has usually been reserved for optimizations which are beyond the then current state of the*

compiler art [Bal85]. The ultimate goal in AP is to automatically build a program from natural language commands, although some critics deem it impossible [RW88, Dij89].

- **Genetic Programming** refers to the usage of Genetic Algorithms to build programs. In GP, a population of programs evolves by reproducing with an appropriate crossover operation and the survival of the most adequate programs is guaranteed by natural using a fitness function [Koz94].

- **Genetic Improvement** is the evolution of existing software using automated search techniques. Evolutionary algorithms and GP are commonly used approaches in current GI solutions [PHH+18].

- **Fault** is the cause of a software **error**, which corresponds to a deviation from the correct state of a program that leads to the observation of a **failure**, a display of incorrect behavior [ALRL04]. For the purposes of this document, the terms **bug** and **fault** will be used interchangeably to refer to what are, according to the definitions of Avizienis et al., *human-made, nonmalicious software faults*.

- **Automated Program Repair**, Automatic Software Repair, Automatic Bug Fixing, among other variants, are all designations used to describe the same concept. This document uses the term **Automated Program Repair** and follows the definition by Monperrus, which states that *"Automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one according to a specification"* [Mon18].

2.1.2 Live Programming

- **Liveness** as a characteristic of a programming environment relates to its ability to provide live feedback to the developer [Tan13]. Tanimoto defines six different levels of liveness, according to the system's behavior. In levels 1-4, the system runs the program and responds to the programmer's actions [Tan90]. In levels 5 and 6, the system also offers predictive capabilities, in the form of action suggestions to the programmer [Tan13].

- **Live Programming** enables programmers to eliminate the delay that is intrinsic to the edit-compile-run (or edit-compile-link-run, depending on the system in question) feedback loop. In it, instead of the 3 phases, there is a single phase in which the programmer is editing the source code and the system automatically runs the program, with live feedback, at varying degrees, as depicted in Figure 2.1, being provided instantly [Tan13]. *Aguiar et al.* argue this concept can be extended to the entire Software Development life cycle, as **Live Software Development** [ARC+19].

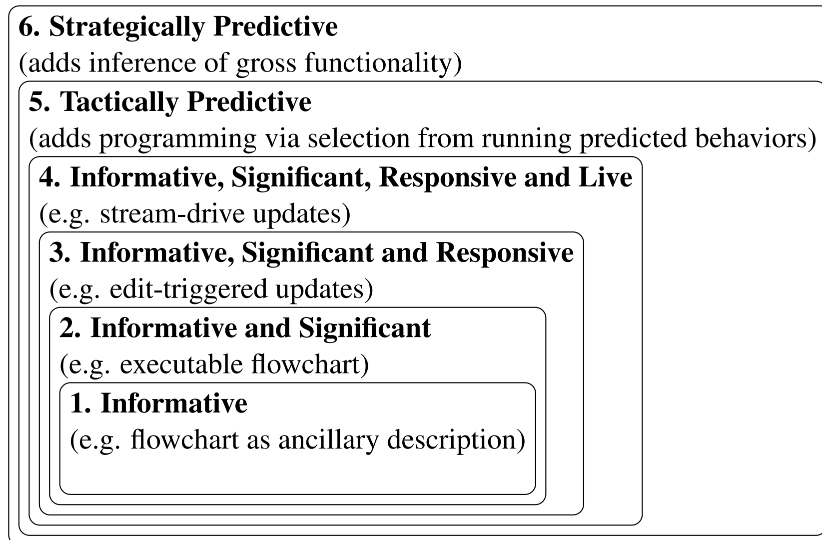

Figure 2.1: Liveness hierarchy as proposed by Steven Tanimoto [Tan13].

2.2 Automated Program Repair

This section describes and compares the state of the art techniques in Automated Program Repair in regards to their chosen approach, effectiveness and, in case of those presenting tools, liveness. In this literature review, only *human-made, nonmalicious software faults*, as defined by Avizienis et al. [ALRL04], are considered as bugs to repair and only general approaches to behavioral repair are evaluated, with solutions targeting a specific class of bugs not being included. For the purposes of this document, a *general approach* — or *generic approach* — is not an approach that guarantees a fix to every bug, but rather an approach targeting more than only one bug class.

In APR approaches, the goal is to attempt to derive a correct program from the faulty one, fixing the error in question without introducing regressions. In order to do so, a specification of what constitutes a correct program is required.

With this specification it is possible to know that a program is faulty, as it does not comply with it, enabling **fault detection**. There are, however, two more steps required in order to repair the program. The second step is **fault localization**, which aims to isolate the code statements responsible for the bug. Finally, this phase enables the final one, the **repair** step.

This survey presents techniques resorting to two kinds of specifications: formal specifications, such as those based on *Design by Contract* [Mey92] programming, and test suites, which can be seen as *informal* specifications for the correct program.

2.2.1 Formal Specifications

A formal specification of a program enables the developer to have a clear representation of what the correct program should be. Multiple authors have leveraged these specifications, in the form of code contracts for instance, and used them as specifications in order to transform a faulty program into a correct one.

2.2.1.1 He and Gupta

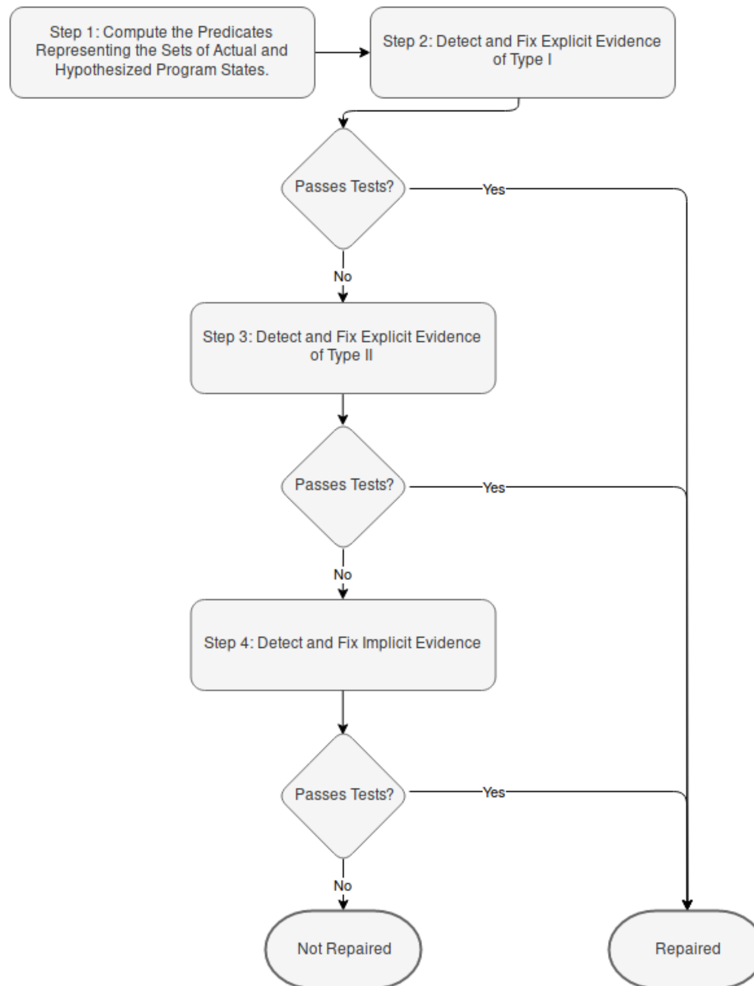

Figure 2.2: Flowchart describing the approach presented by He and Gupta [HG04].

The work presented by *He and Gupta* [HG04] is one of the earliest in the APR field. The solution described leverages postconditions to derive the *hypothesized program state*, corresponding to the expected behavior of the program, which is then compared to the *active program state*, representing the incorrect program. This corresponds to the first of four steps, with the latter three being successive attempts to detect and fix the defect in question, each followed by validation against a test suite, as illustrated by Figure 2.2. In each step, the program is modified by changing either side of an assignment operator or boolean conditions.

This approach is implemented in both C and Python, with experiments ran in 5 different programs and solutions presented to multiple bugs. One drawback of this approach, which the authors acknowledge, is the inability to repair faults present in multiple statements, as it fixes only single statement errors.

2.2.1.2 AutoFix

AutoFix [PFN+14] is an APR technique for the *Eiffel* programming language relying on code contracts, which corresponds to the latest version of an approach first presented in 2010 as **AutoFix-E** [WPF+10]. This solution is reported to fix 42% of 204 faults taking, on average, less than 20 minutes. In addition, 59% of the generated patches are reported to be of high quality, equivalent to human-written ones.

In order to augment the capabilities of the EiffelStudio IDE[1], a plugin was developed that runs in the background and provides suggestions of bug fixes to developers [PFNM15]. By doing so, it achieves a liveness level 5 in the scale proposed by *Tanimoto* [Tan13], by offering live *"tactically predictive"* feedback to the developer.

2.2.1.3 Gopinath et al.

Gopinath et al. [GMK11] develop a technique to repair Java programs making use of the Alloy specification language and corresponding SAT-based tool-set. Their approach relies on the Forge framework for bounded verification [DCJ06] to find the smallest input that violates the postcondition and fault localization techniques to isolate the statement at fault. It then replaces that statement with deterministic behavior with one with non-deterministic behavior, using the specification constraints to force the statement to comply with the postconditions.

This approach is evaluated in two different scenarios - the insert method of a Binary Search Tree and the addChild method of the ANTLR application -, achieving success in both of them, with the bug taking the longest to fix in the BST test requiring 33 seconds and the one in ANTLR 71 seconds.

2.2.1.4 Könighofer and Bloem

Könighofer and Bloem [KB11] present an approach to repair programs in C, requiring both the program and a specification in the form of assertions in the code or as a reference implementation. Based on this, they use symbolic execution [Cla76] and then locate and repair the fault by using template-based repair and SMT solving techniques.

In a later work [KB12], the same authors attempt to perform program analysis during the program repair stage instead of transforming the entire program in a preprocessing step, hypothesizing this would make the process faster. While this hypothesis did not hold true, this approach fixes a larger amount of bugs when compared to the earlier one, when validating against tests instead of an equivalence expression. The method described is also more flexible regarding the type of specification to be used.

[1] https://www.eiffel.com/eiffelstudio/ (Retrieved: 07/02/2019)

2.2.1.5 Rothenberg and Grumberg

The work presented by *Rothenberg and Grumberg* [RG16] describes a mutation-based APR approach resorting to formal specifications. By using formal specifications and targeting a fully complete program, this approach does not resort to the usage of fault localization techniques. It uses a specification in the form of assertions and a set of possible mutations (predefined) to attempt to repair the program. Every repair that is a superset of another successful repair is not considered, guaranteeing the minimal repair. The authors catalogue the mutations into two sets: one containing all possible mutations, named *mutation level 2*, and one subset of that, which enables shorter repair times, named *mutation level 1*.

Table 2.1: Comparison between the works presented in [KB11], [KB12] and [RG16].

KB11		KB12		RG16, mut. lv. 1		RG16, mut. lv. 2	
Fixed	Time (s)	Fixed	Time (s)	Fixed	Time (s)	Fixed	Time (s)
16 (39%)	38	15 (36.6%)	38	11 (26.8%)	2.278	18 (43.9%)	48.151

The results were evaluated on the TCAS program of the Siemens test suite [HFGO94] and compared against the results obtained by *Könighofer and Bloem* [KB11, KB12]. As can be seen in Table 2.1, the approach presented here is able to, in the case of *mutation level 1*, significantly reduce the time needed to repair the bugs in question, although at the cost of repairing a smaller number of bugs or, in the case of *mutation level 2*, repair more bugs, at the cost of a longer repair time.

2.2.2 Tests as Specifications

As test suites are commonplace in industrial and academic settings alike and used as the *validation* for the program correctness, they are often used as specifications to guide the repair of faults in programs. While presenting some difficulties when compared to approaches relying on formal specifications, these methods have the advantage of using a specification that is widely used.

2.2.2.1 GenProg

GenProg [WNLGF09, FNWLG09, LGNFW12] is a method to automatically repair programs using a test-suite as a specification and applying Genetic Programming to fix the bug. In order to do so, this approach requires simply the C program code, a failing test case representing the fault to be repaired and various passing test cases as specification for the correct behavior.

This approach starts by creating an initial population constituted by multiple versions of the starting, faulty, program, with each individual being its AST representation. This first generation is then mutated and evaluated using the fitness function. For each subsequent generation, the crossover operation is applied, followed by the mutation and finally the fitness evaluation, the indicator of the performance of each program. This is repeated until either a program passes all

test cases or a predefined number of generations has passed without a fit repair candidate. After obtaining the *primary repair*, the first individual that passes all test cases, **GenProg** uses program analysis methods to minimize it and obtain the *minimized repair* [FNWLG09].

GenProg provides three possible operators to modify the individuals: addition, deletion and replacement of AST nodes. This process is also built upon the assumption that the possible repair for one bug can be found in the same codebase. This is used in order to reduce the possible search space, by reusing AST nodes from elsewhere in the source code.

Multiple studies have evaluated the performance of **GenProg**, with varying results and debate over them and what constitutes an actual fix. A systematic study by *Le Goues et al.* in 2012 claims that **GenProg** is able to successfully repair 55 out of 105 bugs [LGDVFW12]. The definition of successful repair, as expected by the usage of a test suite as the specification, is dependent on the coverage and quality of test cases. A study by *Smith et al.* found out many of the patches generated by **GenProg** suffer from overfitting to the specific test suite [SBLGB15]. A number of different approaches, such as **Opad** [YZLT17], attempt to minimize the impact of overfitting in APR.

2.2.2.2 Debroy and Wong

The 2010 paper by *Debroy and Wong* [DW10] presents a mutation-based technique to repair programs, using a test suite as specification. It is implemented in both C and Java languages. For this approach, two classes of mutations are considered:

- Replacement of an arithmetic, relational, logical, increment/decrement, or assignment operator by another operator from the same class;
- Decision negation in an if or while statement.

This approach combines the Tarantula [JH05] fault localization technique to rank statements by suspiciousness with the mutation operators that are applied to those statements in order until the correct program is found. It is then evaluated on seven benchmark programs of the Siemens test suite [HFGO94], as well as on the Ant Java build tool[2]. Out of the 135 total bugs (129 from the Siemens test suite and 6 from Ant), this approach is able to successfully repair 25.

2.2.2.3 SemFix

SemFix [NQRC13] is another approach to APR using a test suite as specification, although it uses a **semantics-based** approach, as opposed to the **search-based** ones used in **GenProg** [LGNFW12] and the mutation-based technique presented by *Debroy and Wong* [DW10]. While the latter approaches generate a number of possible solutions and employ search algorithms to find the correct one, this approach extracts a specification from the faulty statement and synthesizes the correct expression according to it.

In order to do so, SemFix follows these three steps:

[2] https://ant.apache.org/ (Retrieved: 07/02/2019)

- **Fault isolation** using Tarantula [JH05], looking at one statement at a time;
- **Statement-level specification inference** inferring the correct specification for that statement, using KLEE [CDE+08] to generate repair constraints;
- **Program synthesis** using the Z3 SMT solver [DMB08] to solve the generated constraints.

The only repairs considered by SemFix are the ones changing the right-hand side of an assignment operator or branch predicates, as it is enough to fix common programming bugs [NQRC13].

SemFix is tested on both the Software-artifact Infrastructure Repository [HFGO94] and some of GNU CoreUtils bugs, with the results being compared to **GenProg**. **SemFix** is able to outperform **GenProg** both in terms of repair time and by fixing more bugs in every program except SIR's *Schedule2*, which include multiple missing-code bugs that **SemFix** is unable to fix with single-statement repairs. This is the main drawback of this approach, as with the growing complexity of software systems, it's not uncommon for errors to be caused by faults in multiple statements.

2.2.2.4 Angelix

Taking into account the *one-statement-at-a-time* limitation intrinsic to **SemFix** [NQRC13], the same group proposed a different approach, named **Angelix**, which aims to scale the repair capabilities to handle multiple faulty statements [MYR16].

This approach, in every aspect similar to the one it is based on, **SemFix**, optimized the symbolic execution phase to support multiple statements, generating an *Angelic Forest* [MYR16], based on the notion of *Angelic Value* [CTBB11] that is also used on their previous work.

By doing this, **Angelix** is capable of fixing a larger number of bugs than **SemFix**, while also providing higher quality patches that introduce less regressions than **GenProg** and other *search-based* approaches.

2.2.2.5 MUT-APR

In a paper comparing approaches reusing existing code and operator mutations, *Assiri and Bieman* present **MUT-APR**, a prototype tool implementing the latter [AB14]. As **GenProg**, it uses genetic programming to search for a solution, differing only in the mutation step, by replacing operators by others within the same class instead of sourcing AST nodes from a different part of the source code. The operator classes supported by this approach are:

- Relational Operators;
- Arithmetic Operators;
- Bitwise Operators;
- Shift Operators.

This approach is then evaluated and compared to **GenProg** using a subset of the Siemens suite [HFGO94], containing only programs with operator faults. Subsequentially, the number of faulty versions was augmented by injecting additional operator mutations. Table 2.2 presents a comparison between the number of repairs found by **GenProg**, 17 out of 54 (31.48%), and by **MUT-APR**, 47 out of 54 (87.03%).

Table 2.2: Comparison between number of patches generated by GenProg and MUT-APR [AB14].

Program	tcas	replace	schedule2	tot_info
Total # of faults	14	21	8	11
MUT-APR	14	18	6	9
GenProg.v2	10	2	5	0

In terms of repair correctness, the study shows that only 27.36% of repairs generated by **MUT-APR** failed regression tests, while for **GenProg** that number is of 96.94%. Additionally, *Assiri and Bieman* argue that repairs done by replacing operators are akin to those produced by developers, and therefore should not reduce maintainability.

2.2.2.6 RSRepair

While various techniques have focused on using Genetic Programming algorithms to evolve software, **RSRepair**, uses Random Search to guide the repair, presenting a modified version of GenProg, but maintaining part of the implementation in steps such as *fault localization* and *mutation* [QML+14].

In the experiments comparing **RSRepair** to **GenProg**, it was able to find repairs to programs faster, while also requiring fewer executions of the test cases. This is evidence that Genetic Programming approaches do not provide a benefit over Random Search to justify the overhead in time required to find a solution.

2.2.2.7 SPR

SPR is an APR approach combining *staged program repair*, a technique to generate repair search spaces efficiently and providing useful repair candidates, and *condition synthesis* generating conditions helping guide the search in the aforementioned search space [LR15].

This approach generated more plausible patches when compared to **GenProg** in the same benchmark as the 2012 study by *Le Goues et al.* [LGDVFW12][3]. In addition, a significant amount of correct patches were also found, while most patches found by **GenProg**, while plausible, do not actually correspond to correct programs.

[3] Out of the 105 reported defects, only 69 are actual defects, with the remaining 36 being deliberate functionality changes [LR15].

2.2.2.8 Prophet

Proposed by *Long and Rinard*, **Prophet** is system leveraging past commits of patches written by human developers to learn a probabilistic model to rank candidate solutions and guide the search for the correct program [LR16].

Evaluated on the same benchmark as the 2012 study by *Le Goues et al.* [LGDVFW12], Prophet is able to generate a plausible patch to 39 out of 69 defects, an improvement when compared to the 36 found by **SPR** [LR15] and the 16 found by **GenProg** [LGNFW12].

Although plausible patches pass all the test cases, these are not necessarily correct patches. In Table 2.3, a detailed comparison of the results reported by *Long and Rinard* can be found. It is clear that, besides generating more plausible patches than **GenProg**, this approach generates significantly more correct patches, indicating a more realistic path towards APR.

On average, it takes **Prophet** 108.9 minutes to get the first plausible patch and 138.5 minutes to get the first correct patch, in the situations in which the first validated patch is correct.

Table 2.3: Comparison between patches generated by GenProg, SPR and Prophet [LR16].

Plausible Patches			Correct Patches		
GenProg	SPR	Prophet	GenProg	SPR	Prophet
16	38	39	2	16,11	18,15

2.2.3 Current Challenges

The current state of the art approaches to Automated Program Repair display promising results, although presenting various limitations and challenges.

Although achieving some degree of success, techniques based on formal specifications are limited by the lack of usage of formal methods and specifications in the industry. Despite a growing number of software developers using formal methods, such as some AWS teams [NRZ$^+$15], it's still not used widely enough to enable Automated Program Repair tools based on formal specifications at scale.

Regarding the techniques based on test suites, the problem of overfitting discussed by *Smith et al.* [SBLGB15] relates to how test suites as informal specifications for the behavior of the program are usually incomplete. Techniques such as **GenProg** have the tendency to overfit, and in spite of solutions such as **Opad** [YZLT17] attempting to eliminate overfitted patches, there's still work to do on either improving the APR algorithms or guaranteeing better test coverage and completeness in terms of specification.

A different, yet also relevant, problem is the lack of readability and maintainability of automatically generated patches [FLW12]. If a software developer is going to be maintaining the program, those are important characteristics which are not enforced by APR approaches.

In regards to the **liveness** of current APR techniques, most solutions do not provide live feedback to the developer. A thorough search of the literature revealed only two approaches, both

providing integration of program repair techniques with IDEs. The first, not covered by this survey of the state of the art due to not fitting its scope, is the one proposed by *Logozzo and Ball* [LB12], which generates repairs for specific warnings produced by *cccheck*. The second, using *Design by Contract* concepts for the specification of the program is **AutoFix** plugin for Eiffel [PFNM15], as described in Section 2.2.1.2. Both these solutions achieve a liveness level 5 in the scale proposed by *Tanimoto* [Tan13], offering *"tactically predictive"* live feedback.

Chapter 3

Problem Statement

3.1 Context and Problem Definition . . . 17
3.2 Research Questions . . . 19
3.3 Validation . . . 19
3.4 Conclusions . . . 20

This chapter describes the problem in detail. Section 3.1 provides the context and problem definition, while Section 3.2 presents the Research Questions to answer. The validation methodology is laid out in Section 3.3. Finally, Section 3.4 summarises this chapter's contents.

3.1 Context and Problem Definition

3.1.1 Automated Program Repair

Automated Program Repair approaches, while displaying encouraging results, still suffer from limitations and present numerous challenges that prevent them from being widely applied in daily use and industrial settings. Among them, as identified in Section 2.2.3, are both the possibility of overfitting to an incomplete specification and the lack of maintainability of automatically generated patches.

As for the first, it pertains to how overfitting to the specific set of tests might break untested behavior [SBLGB15]. As unit tests are rarely *up-to-date* with the source code [RFS+17], the usage of test coverage metrics is not a certainty [GZ13], and comprehensiveness of test suites is rare [LGFW13], it is important to consider that the test suite might not represent the complete specification, thus highlighting the relevance of tackling the overfitting problem in Automated Program Repair systems.

In regards to maintainability, a characteristic we often aim to preserve in codebases we intend to work on in the future, it is not always guaranteed to be enforced by Automated Program Repair

solutions and research points to a lack of maintainability of automatically generated patches when compared to those written by humans [FLW12].

3.1.2 Editor Services

Most IDEs and Text Editors, provide, either natively or through plugins, multiple editor services aiming to help developers while writing code [OVH+17]. These include, but are not limited to, syntax highlighting and checking, code completion, automatic formatting, code navigation, or automatic refactoring, as exemplified by the Python extension for Visual Studio Code[1].

Code completion tools, in particular, display a level 5 in the liveness hierarchy, providing "tactically predictive" feedback by offering program variants to the developer that might mimic what would be the programmer's next changes to the codebase [Tan13]. By doing so, it's possible to speed up development and increase productivity [BMM09]. These tools have, however, largely been limited to *syntactic* suggestions, restricting the value presented to programmers [LBF+12].

As Automated Program Repair techniques obtain better results, true *semantic* suggestions can be offered to developers. Despite the recent success of such techniques, their usage has been predominantly restricted to academically oriented applications, with *AutoFix* [PFNM15] and a tool based on warnings produced by *cccheck* [LB12] being the only tools integrated with IDEs and providing automatically generated patches as live feedback. As discussed in Section 2.2.3, both these tools rely on *Design by Contract* concepts, with the literature review not having revealed any live implementation of an APR tool using test suites as specifications. Furthermore, only one of them - *AutoFix* - offers a general approach to behavioral repair, as defined in Section 2.2.

3.1.3 Problem Definition

With the current limitations of code completion tools in offering exclusively *syntactic* suggestions to developers and the small number of Automated Program Repair tools integrated with IDEs and Text Editors, it is logical to ask what is the possible impact of a *semantic* code completion tool suggesting patches generated by APR techniques.

In order to understand the usefulness of such a tool as a means to improve developer speed and the significance of the overfitting and lack of maintainability issues in this context, the impact should be measured in terms of the speed of the developers as well as the final result of code solutions.

The two APR tools displaying liveness at level 5 (*"tactically predictive"*) that were identified rely on *Design by Contract* concepts as specifications. As such approaches are not the norm in most software engineering contexts, but rather an exception, it is of value to consider relying on more common specifications. One possibility is to use *unit tests* as program specifications, for which the literature review carried out during this dissertation revealed no existing tools with characteristics corresponding to at least a level 5 in the liveness scale.

[1] https://marketplace.visualstudio.com/items?itemName=ms-python.python (Retrieved: 23/06/2019)

3.2 Research Questions

In tackling the problem of understanding the impact of a live Automated Program Repair tool, we aim to find if, by using such a tool, developers are faster in repairing faulty programs and if they attempt to validate or modify the automatically generate patches, as doing so could mitigate the aforementioned overfitting and lack of maintainability problems.

As such, we propose the following hypothesis:

Hypothesis: Using a live Automated Program Repair tool improves the speed and final result of code solutions.

Furthermore, we set the following Research Questions, which we intend to answer in this dissertation:

- ***RQ1:*** *Are users faster in reaching the solution when using a live Automatic Program Repair tool?*
- ***RQ2:*** *Are solutions generated by an Automatic Program Repair tool different from the ones developed by human programmers?*
- ***RQ3:*** *Are users aware of the rationale of solutions generated by the Automatic Program Repair tool before accepting them?*

3.3 Validation

In order to validate the proposed hypothesis and answer the Research Questions, an empirical test was undertaken, in which two groups of programmers performed a series of tasks split into two different problem sets, each containing 4 different tasks. This test was split into two parts and was conducted as follows:

- In the first part:
  - group A attempted to repair problem set X with the live APR tool resulting from this work;
  - group B attempted to repair problem set X without the live APR tool resulting from this work.
- In the second part:
  - group B attempted to repair problem set Y with the live APR tool resulting from this work;
  - group A attempted to repair problem set Y without the live APR tool resulting from this work.

The time taken to solve each problem, as well as information on how test participants interact with the live Automated Program Repair tool and the final code were all recorded, providing the data points to answer the Research Questions of this dissertation.

A thorough account of the Validation process, including the methodology, goals, and a discussion of the results obtained can be found in Chapter 5.

3.4 Conclusions

In order to allow IDEs to provide *semantic* code suggestions in addition to *syntactic* ones, we bridge the Automated Program Repair and Liveness areas of research and understand how a live APR tool impacts the speed and final result of code solutions to faulty programs.

As part of this work, we developed a live Automated Program Repair tool to be used in an empirical test where we compare the performance of programmers while using it and without it, allowing us to gather the data required to answer our Research Questions and test our hypothesis.

Chapter 4

Proposed Solution

4.1 Overview 21
4.2 Automated Program Repair 23
4.3 Visual Studio Code Extension 27
4.4 Summary 30

This chapter presents a description of the implementation, containing both a general overview in Section 4.1 and a more detailed account in Sections 4.2 and 4.3. To close the chapter, Section 4.4 outlines its contents.

4.1 Overview

In Chapter 3 we pointed out the lack of *semantic* suggestions in code completion tools, restricting their potential in improving developer productivity. Owing to the recent advances in the Automated Program Repair space, we put forward the possibility of taking advantage of those techniques to fill that gap and offer true *semantic* suggestions. In addition, we propose that this solution is able to mitigate two of the issues of current APR approaches by suggesting the patches to developers and allowing them to take action accordingly, discarding or modifying the ones that they recognize as overfitting the test suite or presenting severe maintainability issues.

Due to the prevalence of unit tests in the industry and software development world in general, using them as specifications for the automated repair of programs enables us to understand the potentiality of this solution in that context.

To summarise, the proposal is to **leverage unit tests to provide semantic suggestions to developers** as an extension to an existing IDE or text editor. Consequently, it is a prerequisite for the functions to repair have a matching test suite, which is adopted as the specification for the correct program.

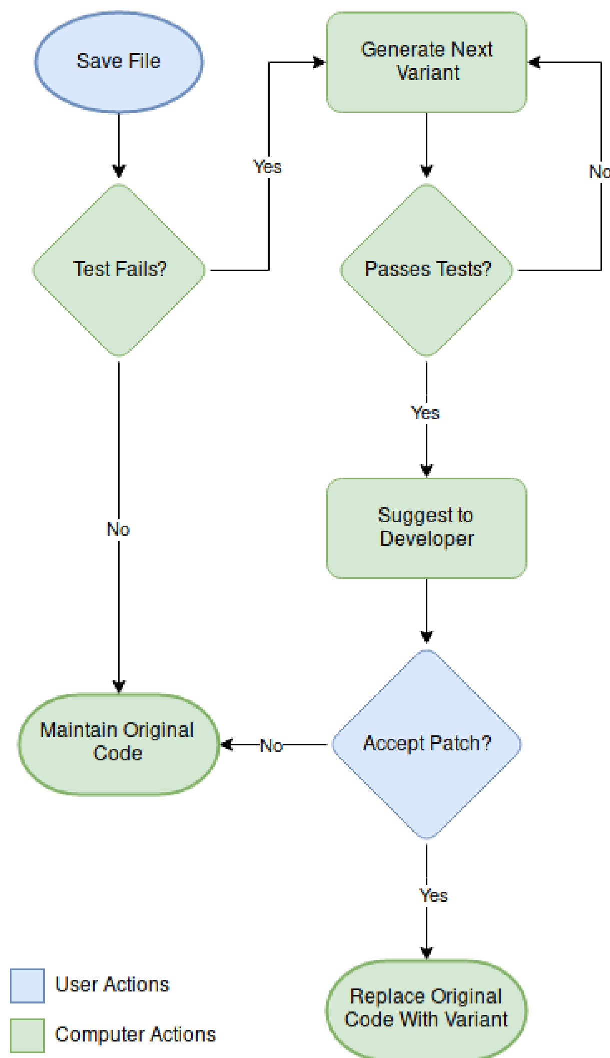

Figure 4.1: Flowchart describing the proposed solution.

In figure 4.1, an overview of the proposed solution is depicted. At any time while coding, by saving the file, the developer launches the automated program repair process, which generates variants for the functions for each at least one of test case fails. Each of these variants is then tested against the test suite and if one is found that passes all test cases, it is considered to satisfy the requirements and suggested to the developer. This suggestion can then be accepted, if deemed acceptable, with the source code being changed to the repaired version.

While this description is general and can easily be transferred to a multitude of programming languages and development environments, the work presented in this report corresponds to a specific implementation. According to the *Stack Overflow Developer Survey 2019*[1], *JavaScript* is the most popular programming language, being used by 67.8% of the respondents, and *Visual Studio Code* the most popular development environment, being used by 50.7% of the respondents. Because of that, those were selected, respectively, as the language to be supported and perform

[1] https://insights.stackoverflow.com/survey/2019 (Retrieved: 25/06/2019)

repairs on and the text editor to build the extension.

An in-depth description of the implementation of the technique used to generate each variant is given in Section 4.2, while Section 4.3 is focused on the integration with the text editor and interaction with the developer.

4.2 Automated Program Repair

In order to successfully provide valid suggestions of fixes to the faulty code, an Automated Program Repair technique is required and a number of factors were taken into consideration in order to implement it.

- **Unit tests as specifications:** In order to satisfy the requirement of producing repairs solely based on unit tests and without relying on formal or any other kind of informal specifications, a test suite must be used as a specification.
- **Generic approach to behavioral repair:** The APR system should not be restricted to a specific class of bugs, as discussed in Section 4.2.
- **Immediate solution:** Offering liveness as an attribute requires providing immediate feedback to the developer, ergo it is required that the solution is found rapidly.
- **Complete solution:** For a solution to be acceptable it must be complete, with partial solutions (those passing at least one test but not all) and syntactically incorrect programs being rejected.

Based on these criteria and the literature review conducted in Chapter 2, a **mutation-based** solution was adopted, such as the one used by *Debroy and Wong* [DW10]. Firstly, it allows for the use of unit tests as specifications, as it requires nothing from the specification besides the expected output. Secondly, by using a heuristic technique to generate each mutant, it is possible to explicitly define which classes of bugs are fixed, avoiding being restricted to a specific one, as well as limiting the search to a timely enough duration that fits into the live programming requisites. Using a Genetic Programming algorithm similar to *GenProg* [WNLGF09, FNWLG09, LGNFW12], while likely achieving a higher success rate in repairing bugs, would be too slow to be suitable for a live tool. Finally, by demanding that a solution passes all test cases, the requirement of suggesting exclusively *complete solutions* is met.

As for what constitutes an *unit* to be tested by each unit test, this approach assumes that *one unit* corresponds to *one function*, hence mutations will only be generated for the function being tested by the failing method. By doing this, the amount of statements to mutate is drastically reduced and so is the search space.

4.2.1 Mutant Generation

Following the decision regarding the general approach, some specifics needed to be settled. For a mutation-based one, it is required to decide which mutations to pursue, as well as the order in

which they appear. This process took into consideration the prior work in this space as well as the type of actions found in patches for the *Defects4J* dataset [JJE14], one of the leading databases of existing faults.

Considering the paramount condition for a live system is its immediacy, three constraints were set in order to reduce the time taken to generate each mutant:

1. **No mutation should increase complexity:** This discards any mutations adding loops or conditional branches, as well as any other changes increasing cyclomatic complexity.

2. **No additional statements:** Since without any additional information there are potentially infinite statements that can be added [DW10], mutations will not add new statements, such as new method calls or variable instantiations. With the goal of maintaining consistency in mind, the opposing mutation - removing statements -, is also disregarded. In short, this limits the mutations to modifications to existing statements.

3. **One mutation per mutant:** While allowing more than one mutation per mutant is likely to result in more solutions being found, the search space rapidly grows when combining mutations, hence keeping the solution restricted to a single mutation per mutant significantly lowers the time required to go through the search space.

The following analysis of the *Defects4J* dataset draws heavily from the the *Defects4J Dissection* paper published by *Sobreira et al.* and the corresponding online appendix[2], which classifies the actions of each repair in the dataset by group and type [SDM+18].

Sobreira et al. identify nine repair action groups are identified, each containing at most three types of operations: *addition*, *removal* and *modification* [SDM+18]. To begin with, four of the groups were fully excluded from consideration for the developed tool, for the following reasons:

- **Method Definition:** Since each mutation is only applied inside the function definition, as the unit under test is the function itself, no method definition mutations are considered.

- **Exception:** This group was excluded due to incompatibility with Constraint 2 (*No additional statements.*), as all patches belonging to it either add or remove statements.

- **Type:** Since JavaScript is a dynamically typed language and not a statically typed one, this group does not apply.

- **Variable:** The only *modification* actions are type and modifier changes, which have no equivalent in JavaScript, and replacing variables by other variables or method calls, which presents potentially infinite variations and is therefore discarded.

From the remaining five groups, only *modification* operations were considered. The groups and respective mutations to implement are the following:

[2] http://program-repair.org/defects4j-dissection/ (Retrieved: 25/06/2019)

- **Assignment:** Modifications on the RHS of an assignment statement.
- **Conditional:** Modifications on the RHS the conditional expression statement.
- **Loop:** Modifications on RHS of initialization variables and modifications on the RHS of conditional test.
- **Method Call:** Method call moving and modifications on parameter value.
- **Return:** Modifications in return expression.

In order to achieve these modifications, five classes of mutations are applied: *Switch Mutations*, *Parentheses Mutations*, *Off-By-One Mutations*, *Operator Mutations* and *Statement Moving Mutations*. The five sections from 4.2.1.1 to 4.2.1.5 describe how each of these mutations is performed. In respect to the five groups identified in the *Defects4J* dataset, the first four mutation classes apply to all of the groups, providing modifications on expressions and call parameters, while the last one applies only to the *Method Call* group, specifically in regards to *method call moving*.

For the purposes of simplicity and clarity, each algorithm is presented here as a pseudocode adaptation of the developed TypeScript code. The proposed approach performs these mutations in two steps. Initially, the algorithm traverses the AST, generating mutations when possible, using the first four classes of mutations as identified above. For each mutation, the node types it applies to will be identified in the respective section. Afterwards, the algorithm iterates through each line of code and switches it with the following line, resulting in all the possible *Statement Moving Mutations*.

The AST traversal and mutations resulting from its manipulation were implemented by using the TypeScript Compiler API[3]. This API provides multiple tools to interact with TypeScript and, by extension, since it is a superset of JavaScript, with JavaScript. The usage of this API and parts of the developed tool is adapted to be used as part of the algorithm pseudocode as follows:

- **Node:** Represents a node in the AST, being the superclass from which the specific node type classes inherit, such as *Binary Expression*.
- **Binary Expression (Node Type):** Represents a Binary Expression node, which has three children: LHS, Operator and RHS. In the pseudocode algorithms, accessing those elements is done by using the properties *lhs*, *op* and *rhs*, respectively.
- **Replacement:** Internal representation of a mutant. Stores both the old and the newly generated node. Creating a new replacement in the pseudocode below is done by *"new Replacement(oldNode, newNode)"* or *"new Replacement(oldFunction, newFunction)"*.

[3] https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API (Retrieved: 25/06/2019)

4.2.1.1 Switch Mutations

The *Switch Mutation* consists in switching the left-hand side and right-hand side of a binary expression, maintaining the operator, as presented in Algorithm 1. This mutation is applied to every Binary Expression.

Algorithm 1 Generate Switch Mutants

```
1: procedure GENERATESWITCHVARIANTS(node, replacementList)
2:     newNode ← new BINARYEXP(node.rhs, node.op, node.lhs)
3:     replacement ← new REPLACEMENT(node, newNode)
4:     replacementList.push(replacement)
```

4.2.1.2 Parentheses Mutations

The *Parentheses Mutation* consists in adding parentheses to both the left-hand side and right-hand side of a binary expression, maintaining the operator, as presented in Algorithm 2. This mutation is applied to every Binary Expression.

Algorithm 2 Generate Parentheses Mutants

```
1: procedure GENERATEPARENTHESESVARIANTS(node, replacementList)
2:     newLhs ← ADDPARENTHESES(node.lhs)
3:     newRhs ← ADDPARENTHESES(node.rhs)
4:
5:     lhsVariantNode ← new BINARYEXP(newLhs, node.op, node.rhs)
6:     rhsVariantNode ← new BINARYEXP(node.lhs, node.op, newRhs)
7:
8:     lhsReplacement ← new REPLACEMENT(node, lhsVariant)
9:     rhsReplacement ← new REPLACEMENT(node, rhsVariant)
10:
11:    replacementList.push(lhsReplacement)
12:    replacementList.push(rhsReplacement)
```

4.2.1.3 Off-By-One Mutations

The *Off-By-One Mutation* consists in generating two variants to each Binary Expression as displayed in Algorithm 3. In both of them, the right-hand side of the expression is modified, firstly by adding one and then by subtracting one, always maintaining both the operator and the left-hand-side.

4.2.1.4 Operator Mutations

The *Operator Mutation*, exhibited in Algorithm 4 as a simplified version, consists in generating one mutant for each other operator of the same class as the original one. In the approach presented in this dissertation, this mutation is applied to every Binary Expression node which includes either

Algorithm 3 Generate Off-By-One Mutants

```
1:  procedure GENERATEOFFBYONEVARIANTS(node, replacementList)
2:      minusOneRhs ← node.rhs − 1
3:      minusOneNode ← new BINARYEXP(node.lhs, node.op, minusOneRhs)
4:      minusOneReplacement ← new REPLACEMENT(node, minusOneNode)
5:
6:      plusOneRhs ← node.rhs + 1
7:      plusOneNode ← new BINARYEXP(node.lhs, node.op, plusOneRhs)
8:      plusOneReplacement ← new REPLACEMENT(node, plusOneNode)
9:
10:     replacementList.push(minusOneReplacement)
11:     replacementList.push(plusOneReplacement)
```

an *arithmetic* or a *relational* operator. For every mutant, the left-hand side and right-hand side of the expression are kept unchanged.

Algorithm 4 Generate Operator Mutants

```
1:  procedure GENERATEOPERATORVARIANTS(node, replacementList)
2:      for newOp in OtherOperatorsList do
3:          newNode ← new BINARYEXP(node.lhs, newOp, node.rhs)
4:          replacement ← new REPLACEMENT(node, newNode)
5:          replacementList.push(replacement)
```

4.2.1.5 Statement Moving Mutations

The *Statement Moving Mutation*, exhibited in Algorithm 5, is fundamentally different from the previous four mutations. Instead of manipulating the AST, it works directly on the source code, by switching each pair of adjacent lines among them.

Algorithm 5 Generate Statement Moving Mutants

```
1:  procedure GENERATESTATEMENTMOVINGVARIANTS(code, replacementList)
2:      linesList ← code.getLines()
3:      for i ← 0 to linesList.length − 1 do
4:          newCode ← SWITCHLINES(code, i, i+1)
5:          replacement ← new REPLACEMENT(code, newCode)
6:          replacementList.push(replacement)
```

4.3 Visual Studio Code Extension

In order to provide the program repairs as suggestions, interaction with the developer through the text editor is required. The tool resulting from this work requires three capabilities to offer live feedback:

- **Accept User Input:** As defined previously, the entire APR process takes place every time the developer saves the file he's currently working on.
- **Display Suggestion:** The identified bug fixes should be displayed to the developer both by pointing out the location and display the proposed code changes.
- **Apply Changes:** The candidate solutions, if accepted by the developer, should replace the previous code.

Through the *Visual Studio Code Extension API*[4], hereinafter the *Extension API*, it is possible to customize and extend most of the editor's functionalities. In Sections 4.3.1 through 4.3.3, the usage of this API in implementing the three required capabilites is explained.

4.3.1 Accept User Input

The *Extension API* provides programmers with access to multiple events within the Workspace[5], a representation of the folder that is currently open in the editor. The extension implemented for this dissertation relies on two of them to prompt the APR process: one that triggers whenever a text document is opened - *workspace.onDidOpenTextDocument* -, and another whenever one is saved - *workspace.onDidSaveTextDocument*. This means that, besides satisfying the requirement of running the APR system whenever the user saves a text document, it runs when first opening it as well.

4.3.2 Display Suggestion

Displaying suggestions to the developers is in itself comprised of two problems: pointing out the location of the fix to the developer when the editor is open and actually displaying the fix as a *semantic* suggestion. Because of this, the implemented tool works more akin to automatic refactoring tools than traditional code completion ones.

```
20  function diagonalDifference(matrix) {
21      let len = matrix.length;
22      let firstDiag = 0;
23      let secondDiag = 0;
24
25      for (let i = 0; i < len; i++) {
26          firstDiag = firstDiag + matrix[i][i];
27          secondDiag = secondDiag + matrix[len - i][i];
28      }
29
30      return Math.abs(firstDiag - secondDiag);
31  };
```

Figure 4.2: Underlined bug fix location.

[4] https://code.visualstudio.com/api (Retrieved: 25/06/2019)

[5] https://code.visualstudio.com/api/references/vscode-api#workspace (Retrieved: 25/06/2019)

In order to point out the location of the fix, the implementation makes use of Diagnostics[6]. This underlines the specified location with the color of the desired severity level. For the purpose of this extension, as pictured in Figure 4.2, it was decided to attribute to bug fixes the severity of *Error*, with the underline being drawn in red, as a fault in the program can be considered an error.

The second part of the problem, also resorts to the Diagnostic object, by setting the message with the proposed code change. In Figure 4.3, an example of the dialog box displayed when hovering over the fix location is given. The suggestion is given as a text message in the form *Replace: current code ==> suggested code*.

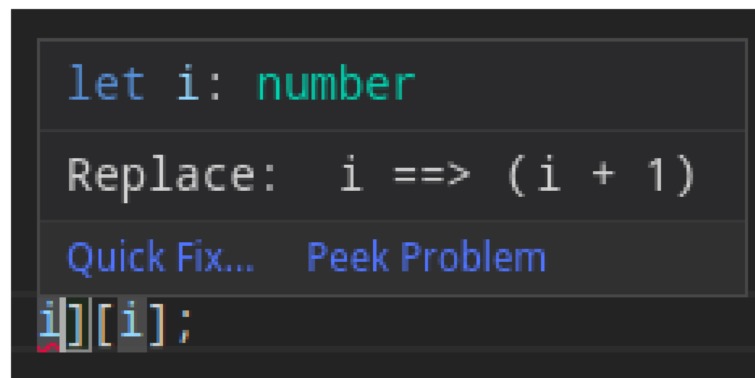

Figure 4.3: Example of a patch suggestion in a dialog box.

In addition, if the developer has the *Problems* view open, the message appears there, drawing attention to the bug and corresponding fix. By clicking the specific problem, the focus shifts to the bug location, allowing the developer to then make the most appropriate decision.

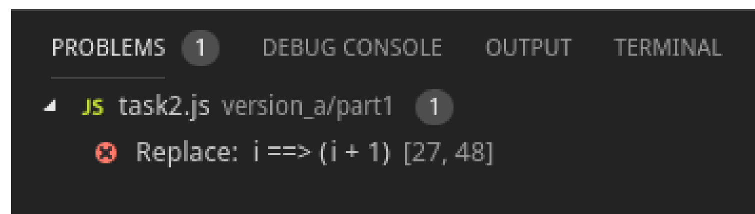

Figure 4.4: Example of a patch suggestion in the problems tab.

By running the tool while the developer is coding and offering these suggestions immediately, the extension described in this chapter achieves a level 5 in the liveness scale. In order to fully automate the *semantic* suggestions approach described in Section 4.1, however, one last step is required.

[6] https://code.visualstudio.com/api/references/vscode-api#Diagnostic (Retrieved: 25/06/2019)

4.3.3 Apply Changes

The final capability the extension needs to support is that of applying the changes directly to the source code, in case the user accepts them. To do so, the *CodeActionProvider*[7] functionality of the *Extension API*, an interface for code actions, was implemented in a class responsible for providing these suggestions as automatic changes.

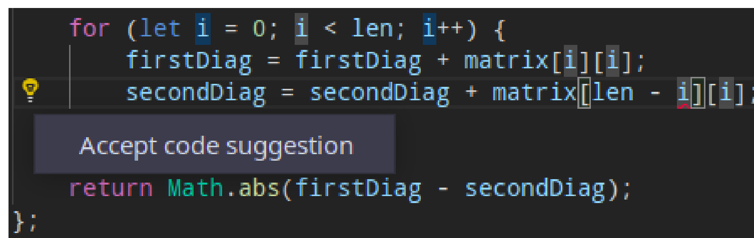

Figure 4.5: Example of the *light bulb* with code actions.

As can be seen in Figure 4.5, when a solution is found, a yellow light bulb appears to the left of the code in the same line of the located fault. If the developer wants to, simply clicking on *Accept code suggestion* will update the source code. However, if the current source code does not match the source code the suggestion engine has as the *"old code"* (the left-hand side of the *Replace: current code ==> suggested code* message), the extension will not apply the changes and will instead display an error message.

4.4 Summary

This chapter described the proposed solution and its implementation in detail. In Section 4.1 a general overview of the solution, agnostic from this specific implementation, was given. It was also defined that the programming language to target would be *JavaScript* and the text editor to build the extension for would be *Visual Studio Code*.

In Section 4.2, the approach to repair the program faults was presented, with a mutation-based one being selected. After that, the specific mutation operations were described, as well as the reasoning behind the decisions taken in defining them.

Finally, the integration with Visual Studio Code, by using the *Visual Studio Code Extension API* was described both from the viewpoint of the user and by describing the API functionalities used during development.

[7] https://code.visualstudio.com/api/references/vscode-api#CodeActionProvider (Retrieved: 25/06/2019)

Chapter 5

Empirical Evaluation

5.1 Methodology **31**
5.2 Results **35**
5.3 Threats to Validity **47**
5.4 Summary **49**

This chapter presents an account of the empirical evaluation of the developed tool. Firstly, the methodology followed in this study is described in Section 5.1. Then, in Section 5.2, the results, respective analysis and a brief discussion are presented, with the threats to validity being enumerated and discussed in Section 5.3. Lastly, Section 5.4 summarises the contents of this chapter.

5.1 Methodology

Experimental validation plays a crucial role in science, enabling us to validate the proposed hypothesis and it should be no different in the field of Software Engineering [FPG94], for which empirical work should be considered *first-class science* [TLPH95]. Experimentation is particularly relevant when evaluating the proposed solution requires the implementation of a prototype [DC02].

With this in mind, an empirical study was conducted with the goal of validating the hypothesis proposed in section 3.2. This study consisted in evaluating the performance of developers when using the implemented *Visual Studio Code* extension and comparing it to the performance when not using it, by measuring the time taken to solve each problem, and assessing the form of the final code developed.

5.1.1 Plan

Empirical testing requires establishing the selection criteria for participants and the data to observe, as well as careful planning of the process to follow. In this section, the parameters defining the experiment are described.

Participants. The tests consisted in repairing faulty *JavaScript*, hence software development experience and at least basic familiarity with *JavaScript* was required for all participants. 16 software developers participated in this study.

Duration. The estimated duration for the test was 30-40 minutes. However, and in order to strike an equilibrium between a reasonable time to finish all of the tasks and a sensible time as to not overburden the volunteers, we set a maximum time of one hour to solve the eight tasks. As such, each task had a *timeout* for each task being set at 7 minutes and 30 seconds.

Environment. The experiment was conducted in a room at the Faculty of Engineering of the University of Porto. Every participant used the same computer and development environment: Manjaro with the GNOME desktop environment, *Visual Studio Code* and the tool resulting from this work. During the test, the participants had full access to the Internet.

Procedure. In order to be able to compare the performance of developers using the developed *Visual Studio Code* extension and without using it, the 16 participants were split into two groups of eight participants each, *group A* and *group B*. As no guarantees of equality of technical knowledge between the two groups exist *a priori*, the need for both to be exposed to the extension resulted in the creation of two different but equivalent problem sets. The test was therefore comprised of two different parts, in accordance with the following procedure:

In the first part:

- *group A* attempted to repair *problem set X* with the live APR tool resulting from this work;
- *group B* attempted to repair *problem set X* without the live APR tool resulting from this work.

In the second part:

- *group B* attempted to repair *problem set Y* with the live APR tool resulting from this work;
- *group A* attempted to repair *problem set Y* without the live APR tool resulting from this work.

Data. An empirical evaluation implies observing and collecting data in order to validate the proposed hypothesis. During the experiment, two things were recorded for each task: (1) the time it took to reach a solution, and (2) the final code solution. In addition, for tasks performed while using the live APR tool, whether the participants used it or not was also recorded.

Survey. A survey was carried out with four different parts, each with a different goal:

1. a **background** section which aimed to assess the level of technical knowledge familiarity with the used technologies;
2. **after part 1** of the test, a set of statements pertaining to the problems solved in that part (specific depending on whether the extension was used or not), meant to understand what users thought of the problems and their performance;
3. **after part 2** of the test, the other version, related to the extension's usage or not, had a similar goal;
4. a **post-test** survey, with statements related to the development environment, the overall satisfaction, as well as possible future improvements.

Each part was comprised of a set of statements, with the answers being recorded in a *Likert scale* [Lik32] with five possible responses: *Strongly Disagree*, *Somewhat Disagree*, *Neither Agree nor Disagree*, *Somewhat Agree* or *Strongly Agree*.

5.1.2 Tasks

While a number of benchmarks used to evaluate Automated Program Repair techniques exist[1], the goal of this empirical study is not to evaluate the quality of the repairs produced by the implemented prototype, but rather its usage developers. Because of this, a dataset was created specifically for this study, taking into consideration its scope and objectives.

As defined in Section 4.2, for the purpose of this work, *one unit* is defined as *one function* and for that reason, each problem will correspond to a function. Three different types of problems are considered when using the live APR tool, depending on if and when the extension finds a solution, which we defined as follows:

(i.) **immediate:** functions for which a bug is already present and a solution fixing it is immediately found by the extension.

(ii.) **nonimmediate:** functions for which a bug is already present but a solution fixing it is only found by the extension after a piece of code, corresponding to a missing functionality, is written.

(iii.) **nonpresent:** functions for which a bug is not present, but the participants must write a piece of code, corresponding to a missing functionality, which might contain a bug the extension is able to fix.

As for the problems of type (i.), they are used as *sanity checks*, showing the effectiveness of the tool for the simplest of problems. The other two types allow for the study of the tool in solving more complex problems, with type (ii.) problems being used to understand the effectiveness of the

[1] http://program-repair.org/benchmarks.html Retrieved: 26/06/2019

tool in helping fix bugs not introduced by the developer and those of type (iii.) for bugs directly caused by the developer.

Considering these types and the requirements set in Section 5.1.1, including the need for two different problem sets, we defined that each problem set would be comprised of four questions: two of type (i.), one of type (ii.) and one of type (iii.). The specific tasks for both problem sets are described in this section.

Task 1

The first task was one of the *immediate* problems — type (i.) —, allowing us to validate the extension in terms of its basic usability. For this task, an *operator mutation* is enough to fix the bugs in question.

In **Problem Set X**, we chose to introduce a fault in an implementation of Bubble Sort, by inverting a comparison operator.

In **Problem Set Y**, we opted to use a problem from the */r/dailyprogrammer subreddit*[2], for which a function should find if every letter appearing in the input string does so the same number of times. We adapted one of the *JavaScript* solutions proposed in the comments section. As in *Problem Set X*, the bug is an inverted comparison operator.

Task 2

This second task — of type (i.) — serves the same purpose as the first one, also working as a *sanity check* for the extension. Once again, the program repairs are simple, with an *off-by-one mutation* being sufficient.

In **Problem Set X**, we adapted a popular question used in various programming challenges that, given a matrix, requires the absolute difference between the two diagonals. The bug is solved with an *off-by-one mutation*.

In **Problem Set Y**, we opted to use a problem from the */r/dailyprogrammer subreddit*[3], for which a function calculating the score of a game based on an input string is required. We adapted one of the *JavaScript* solutions proposed in the comments section. The bug is an incorrect operator but can be solved with either an *off-by-one mutation* or an *operator mutation*.

Task 3

The goal of the third task — of type (ii.) — was to assess the behavior of the extension in circumstances in which the bug is present but is not found immediately. As such, both versions of this task have a missing part of the code — a line — and a bug that is already present but in a different line.

[2] https://www.reddit.com/r/dailyprogrammer/comments/afxxca/20190114_challenge_372_easy_perfectly_balanced/ (Retrieved: 28/06/2019)

[3] https://www.reddit.com/r/dailyprogrammer/comments/8jcffg/20180514_challenge_361_easy_tally_program/ (Retrieved: 28/06/2019)

In **Problem Set X**, we opted to use a problem from the */r/dailyprogrammer subreddit*[4], requring a function calculating the reverse factorial of the input number. The notion of reverse factorial is based on that of factorial: If $n! = m$, then n is the reverse factorial of m. We adapted one of the *JavaScript* solutions proposed in the comments section, removing a line of code. The bug is an inverted operator but can be solved with either an *off-by-one mutation* or an *operator mutation*.

In **Problem Set Y**, we opted to use a problem from the */r/dailyprogrammer subreddit*[5], for which a function, given the amount to give and a list of coins, calculates the minimum number of coins required to reach that amount. We adapted one of the *JavaScript* solutions proposed in the comments section, removing a line of code. The bug is an incorrect operator which, due to the characteristics of the specific implementation, can be solved with either an *off-by-one mutation* or an *operator mutation*.

Task 4

The goal of the fourth and final task — of type (iii.) — was to evaluate the extension in cases where the bug is introduced by the developer. Neither version of this task contains a bug, but the functionality must be implemented, with both problems requiring solutions that we believe are prone to *off-by-one* bugs.

In **Problem Set X**, participants were required to implement a function returning the substring between two characters, given a string and two indexes (which should both be included in the substring).

In **Problem Set Y**, a similar problem was posed. In this task, the participants were required to implement a function returning the slice of an array between two elements, given an array and two indexes (which should both be included in the slice).

5.2 Results

This section presents the results of the undertaken study, as well as a brief analysis for each of the individual tasks and parts of the survey.

The statistical methods used to analyze study results have been discussed extensively over the years, with criteria such as the nature of the dependent variable, sample size and distribution of the population often being the source of disagreement and controversy. The data under analysis here has been collected from different sources and is of varying types.

Consensus over analysis of Likert-Type items — individual questions — and scales — sums or averages of multiple items — [CD94] is particularly rare [Bro11]. For the purposes of this study, both items and scales will be analyzed, and a need to choose the methods to do so arose.

[4] https://www.reddit.com/r/dailyprogrammer/comments/55nior/20161003_challenge_286_easy_reverse_factorial/ (Retrieved: 28/06/2019)

[5] https://www.reddit.com/r/dailyprogrammer/comments/7ttiq5/20180129_challenge_349_easy_change_calculator/ (Retrieved: 28/06/2019)

While comparing those methods is beyond the scope of this dissertation, it is of importance to briefly contextualize this discussion, so as to justify their usage. Some authors, such as *Clason and Dormody* [CD94], *Jamieson* [J+04] and *Norman* [Nor10], claim that Likert-Type items are on the *ordinal* measurement scale while others argue that it can be analyzed in the *interval* scale, such as *Brown* [Bro11] and *Willits, Theodori and Luloff* [WTL16]. In practice, as *Brown* [Bro11] states, Likert-Type items have often been analyzed as *interval* in nature. Regarding Likert-Type scales, while disagreement is also prevalent, multiple studies have shown they can be analyzed in the *interval* scale [CP08].

In this study, analysis of Likert-Type scales was performed in the *interval* measurement scale and of Likert-Type items in the *ordinal* scale [BB12].

For the analysis of the test results, three different measurements have to be considered: the time to reach the solution, the solution itself, and whether the person used the extension's suggestion. For the the analysis of the time to reach the solutions, the *Mann–Whitney U (MWU)* test[6] was used, as, for small sample sizes, the relatively small variations in the results in comparison with the *t-test* when the population follows a normal distribution are eclipsed by the large variations in distributions with extreme skews, for which MWU is the most appropriate test [BS99]. Regarding the final code of each answer, analysis at a descriptive level was done, with an emphasis on discussing differences between solutions generated by the extension and those developed by participants not using the extension. The usage of the extension's suggestion is a binary variable and will be treated accordingly.

For all tests of the null hypothesis used throughout this dissertation, the significance level to compare the *p-value* was set at the commonly used value of *0.05*.

5.2.1 Background

The *Background* section of the survey, consisting of multiple Likert-Type items, was designed to be handled as a Likert-Type scale assessing the level of comfort of the participants with the technologies used in the experiment and the type of problems to solve. This scale is comprised of the eight questions of the *Background* section, identified here as *B1* through *B8*:

B1: I have considerable experience with JavaScript.

B2: I have considerable experience with the Mocha testing framework.

B3: I have considerable experience with Visual Studio Code.

B4: I have considerable experience with Test Driven Development.

B5: I regularly use tools to help me code (linters, code completion, etc.).

B6: I am always capable of understanding code I haven't seen before.

B7: I feel comfortable in identifying bugs in code I haven't seen before.

B8: I feel comfortable in fixing bugs in code I haven't seen before.

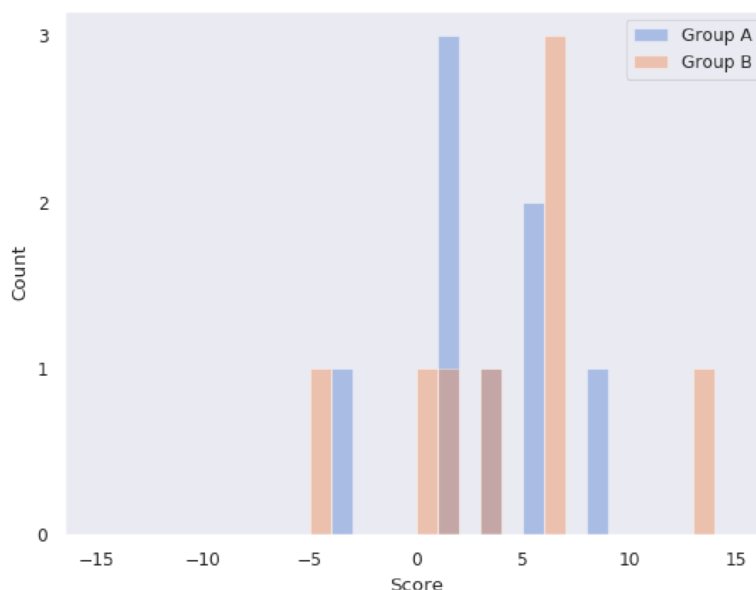

Figure 5.1: Histogram of background scores of both participant groups.

The score of each participant was obtained by summing the answers to each individual item, on a scale of *-2* (*Strongly Disagree*) to *2* (*Strongly Agree*). Owing to this transformation, and since there are 8 items in the *Background* Likert-type scale, the minimum possible score is set at *-16* and the maximum score at *16*, with *0* corresponding to an average one.

As a first step in analyzing the scores, the histograms of both *Group A* and *Group B* were plotted in Figure 5.1, displaying a tendency for scores of both group to skew towards the right, albeit only slightly so. Consequently, the groups appear to be slightly above average in terms of background knowledge regarding the technologies and types of problem present in this study. In addition, despite one specific outlier in *Group B*, no significant difference between the groups seems apparent from these histograms.

Table 5.1: Statistical measures and *p-values* for hypothesis tests on background scores.

Group	Size	Mean	Std. Deviation	Shapiro-Wilk (p)	Levene (p)	t-test (p)
A	8	2.50	3.63	0.69	0.39	0.59
B	8	3.75	5.34	0.80		

As can be seen in Table 5.1, both groups present a slightly above average mean. However, in order to better understand the nature of the groups and whether or not there are statistical differences between their means, further analysis is required. In order to test the null hypothesis that both groups have identical means, we opted to use the *Student's t-test*. This test relies on the assumption

[6]also called *Mann–Whitney–Wilcoxon (MWW)*, *Wilcoxon rank-sum test* or *Wilcoxon–Mann–Whitney test*

of a normal distribution of the population and equal variances, although multiple studies have shown it is fairly robust to violations of at least one assumption [KHL11] and is able to handle small sample sizes [DW13].

In any case, before using the *Student's t-test*, we tested for these assumptions and the *p-values* resulting from each test can be seen in Table 5.1.

Firstly, we used *Levene's test for equality of variances*, which tests the null hypothesis that *both groups are from populations with equal variances* [Lev61]. As the obtained *p-value* is of *0.39*, which is above the previously defined significance level of *0.05*, we can't reject the null hypothesis.

In second place, we used the *Shapiro-Wilk* test to test each of the groups for the null hypothesis that *the sample comes from a population with a normal distribution* [SW65]. Since the resulting *p-value* is well above the significance level of *0.05*, we can't reject the null hypothesis.

As a consequence of these two tests, we assumed that both samples come from normally distributed populations and their variances are equal and proceeded to use the *Student's t-test*. This test enables us to test for the following hypotheses:

$\mathbf{H_0}$ (Null Hypothesis)**:** The means of the populations from which each group was sampled are the same.

$\mathbf{H_1}$ (Alternate Hypothesis)**:** The means of the populations from which each group was sampled are different.

After calculating the *Student's t-test*, we obtain a *p-value* of *0.59*, well above the significance level of *0.05*. Therefore, we fail to reject the Null Hypothesis $\mathbf{H_0}$ and therefore must accept that there is no statistical difference between the two groups.

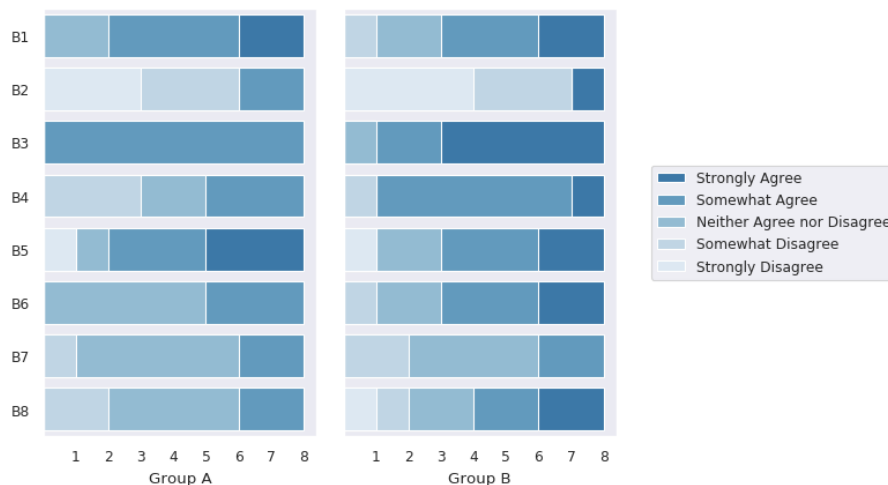

Figure 5.2: Stacked Bar Chart with frequencies of each *Background* survey item.

"Besides the above analysis of the Likert-Type scales, some of the specific items reveal interesting details. For each of those individual items, the frequency of each answer is depicted in Figure 5.2. Of particular note are *B1: I have considerable experience with JavaScript.* and *B3: I have considerable experience with Visual Studio Code.* as in both groups the answer distributions

show a general familiarity with the programming language and the text editor used in this experiment. Conversely, *B2: I have considerable experience with the Mocha testing framework.*[7] is the item which participants tended to disagree the most with, revealing a lack of experience with the testing framework used in this study.

5.2.2 Problem Sets

During the study, each participant was required to solve eight different problems, as described in Section 5.1.2, with *Group A* having used the extension in *Problem Set X* and *Group B* in *Problem Set Y*. From their performance, three different things were recorded:

- the **time** taken to reach a solution;
- the final **code** of the solution;
- if applicable, whether the participant **used the extension's suggestion** or not.

5.2.2.1 Time to Reach Solution

In Table 5.2, we can compare the mean time to reach the solution of the group using the tool against that of the other group, for each of the individual problems. An initial analysis of the table points to a tendency for the group using the tool to be, on average, faster in solving the problem. This difference between the means of both groups is very significant for both tasks of type (i.), tasks 1 and 2, and less so for the tasks of types (ii.) and (iii.), respectively task 3 and 4. These initial observations are according to the expectations, with the tasks of type (i.) being the ones in which the extension provided a bug fix immediately to the participant, while the other two tasks require writing additional code before the extension is able to give any suggestion.

In order to understand whether the difference between the two groups is statistically significant, we opted to do a two-sided *Mann–Whitney U* test for each of the tasks, which tests the following hypotheses:

$\mathbf{H_0}$ (Null Hypothesis)**:** The distributions of the populations from which each group was sampled are identical.

$\mathbf{H_1}$ (Alternate Hypothesis)**:** The distributions of the populations from which each group was sampled are not identical.

As can be seen in Table 5.2, the *p-value* is lower than the significance level of *0.05* for six out of the eight problems in this experiment: both versions of tasks 1 and 2, and tasks 3 and 4 of *Problem Set X*. Therefore, for these 6 tasks, easily identifiable in the table by the underlined *p-value*, we can reject the Null Hypothesis $\mathbf{H_0}$ that the distributions of the populations are identical. For the remaining two tasks, we can't reject the Null Hypothesis $\mathbf{H_0}$ and therefore are unable to evidence statistical differences between the two groups.

[7] `https://mochajs.org/` (Retrieved: 29/06/2019)

Table 5.2: Statistical measures and MWU *p-value* of time to reach solution for each problem.

Task	Set	Tool	Mean	Std. Deviation	Mann-Whitney U *p-value*
1	X	Yes	00:44	01:09	<u>0.00736</u>
		No	02:55	01:59	
	Y	Yes	00:22	00:19	<u>0.00094</u>
		No	03:58	01:51	
2	X	Yes	00:38	00:36	<u>0.00325</u>
		No	03:19	02:13	
	Y	Yes	00:31	00:41	<u>0.00195</u>
		No	03:22	01:16	
3	X	Yes	03:12	00:43	<u>0.01804</u>
		No	04:39	01:21	
	Y	Yes	04:42	01:18	0.20691
		No	05:20	01:06	
4	X	Yes	01:14	00:36	<u>0.01166</u>
		No	02:34	01:04	
	Y	Yes	00:48	00:15	0.27015
		No	01:01	00:24	

5.2.2.2 Code

Due to the nature of the problems, as minimal changes to the code suffice to solve them, analysis of the solution of each participant is unlikely to benefit from the usage of any code quality metrics such as *Cyclomatic Complexity* or *Lines of Code*. Instead, multiple observations were recorded while participants were solving the problems and we looked at the source code of each answer based on these notes. After doing so, various similar situations were found, all of them slight variations of the example described in this section.

In task 3 of *Problem Set Y*, there is a faulty *if condition* that should check whether the sum of the amount of the coins already selected to give as change with the face value of a candidate coin is less than or equal to the desired change value, but in the faulty state does not check for equality.

```
1  if (coin + map.coinTotal < change)
```

The fix suggested by the extension for this problem is the following:

```
1  if ((coin - 1) + map.coinTotal < change)
```

Assuming that coins are always *integer* values, this solution is technically correct, as $n-1<m$ is equivalent to $n <= m$, as long as both n and m are integers. However, one can argue that the latter is a clearly better solution in terms of conveying the *meaning* of the condition:

```
1   if (coin + map.coinTotal <= change)
```

Out of the group that did not have the extension active to solve this problem, every single one of the participants reached this solution, but from the group that did have the possibility to use the extension, only one of the participants opted for this solution, with the remaining seven having accepted the extension's suggestion.

5.2.2.3 Survey

After solving the problems in each of the version, participants were required to fill a survey concerning those problems, with different versions being shown to those who used the extension and those who didn't. The questions are identified by *X1* through *X8* for *Problem Set X* and *Y1* to *Y8* for *Problem Set Y*, and are the same in both versions. These questions are identified here as *P1* through *P8*:

P1: The bugs were easy to identify.

P2: The solutions were straightforward.

P3: I solved every problem correctly.

P4: I spent more time in identifying the bug than in solving it.

P5: The extension was faster in identifying fixes than me.

P6: The extension was able to correctly fix problems.

P7: I used the fixes suggested by the extension.

P8: I tried to understand the fixes suggested by the extension before accepting them.

Of these 8 questions, the first four were asked to both groups, while the others were posed only to the group that used the extension for this problem set.

Problem Set X

Regarding *Problem Set X*, as exhibited by Figure 5.3, there are no major differences in the distribution of answers to X1, *X2* and *X3*, except for a slight skew to the right for *Group B* on *X2*, suggesting a small inclination of those using the extension to agree that the solutions were straightforward. In *X4*, the intergroup variation is notorious, with the group using the extension disagreeing more often than the other group that they spent more time in identifying the bug than in solving it.

In what concerns the questions shown exclusively to the group using the extension — X4 through X8 —, participants were significantly in agreement with the statements we can relate to the usefulness of the extension: *X5*, *X6* and *X7*. The answers to whether the participants tried to understand the suggestions of the extension or not in statement *X8*, while still trending towards agreement, were more dispersed.

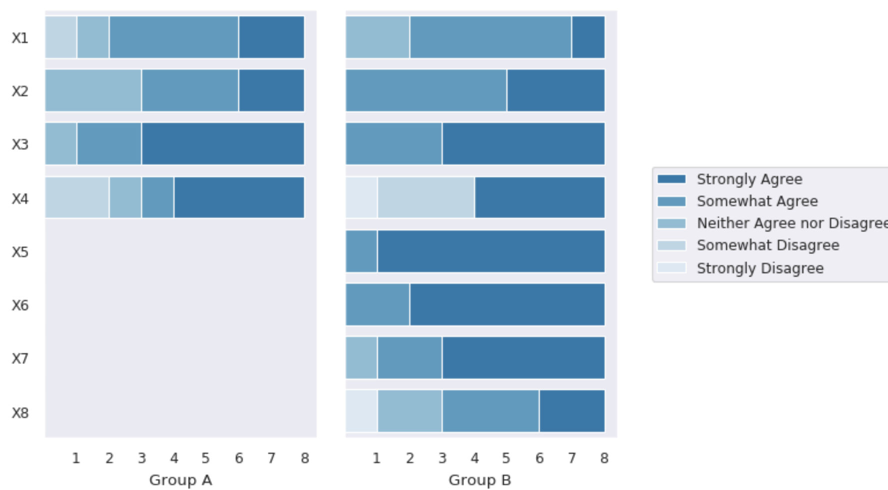

Figure 5.3: Stacked Bar Chart with frequencies of each *Problem Set X* survey item.

Problem Set Y

In *Problem Set Y*, groups displayed more differences in answers to the first four statements (see Figure 5.4). *Group B*, the one not using the extension, was more disagreeing with *Y1* and *Y2*, revealing that identifying the bugs and repairing them was more difficult without the extension. As in *Problem Set X*, the variation in *Y4* was significant, although neither group was overly disagreeing with having *spent more time in identifying the bug than in solving it*.

In statements *Y5*, *Y6* and *Y7*, responses were once again markedly in agreement, as was already the case in *Problem Set X*. Regarding *Y8*, stating that the participant *tried to understand the fixes suggested by the extension before accepting them*, the answers were polarized and no clear tendency towards agreement or disagreement can be identified.

General Remarks

Altogether, the responses to this part of the survey provide us with an insight into how the participants felt with respect to the problems they were asked to solve. Overall, we can identify a very slight tendency to consider the problems easier when using the extension, as evidenced primarily by *X2* (see Figure 5.3), *Y1* and *Y2* (see Figure 5.4). In addition, responses to *X4* and *Y4* point to the participants being less agreeing with having *spent more time in identifying the bug than in solving it*.

Regarding how participants perceived the usefulness of the extension, the consensus was that it was not only able to correctly fix problems, but faster than the participants in doing so. In addition, participants reported having accepted those suggestions often. Whether the participants felt they had a critical attitude towards the extension's suggested fixes or not was a more divisive question, and no clear trend can be identified.

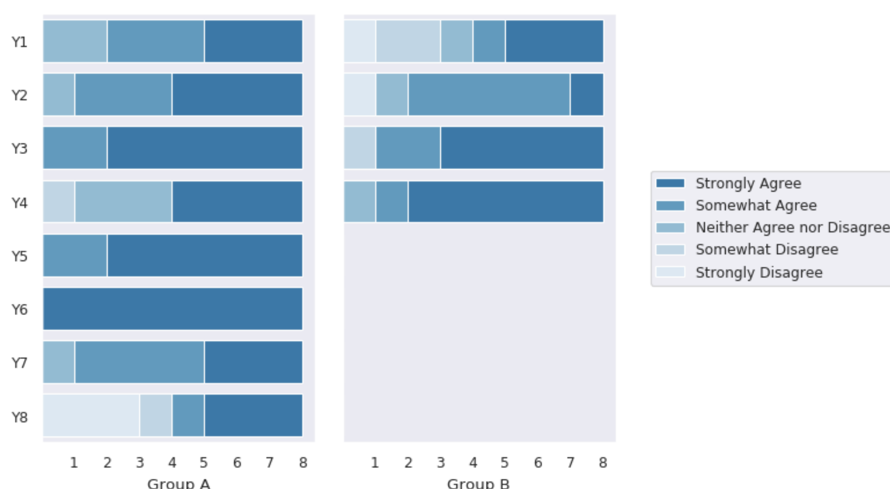

Figure 5.4: Stacked Bar Chart with frequencies of each *Problem Set Y* survey item.

5.2.2.4 Usage of Extension

While each group had the extension active for one of the problem sets, each participant could accept, ignore or discard the fix suggestions. In fact, for tasks 3 and 4, depending on the specific changes the participants made to the code and the order in which they were made, there was no guarantee that a suggestion would even be made. For each task, whether a suggestion was accepted or not was recorded.

Table 5.3: Percentage of participants that accepted a fix suggestion per problem.

Task	Version	Usage (%)
1	X	87.5
	Y	87.5
2	X	87.5
	Y	100.0
3	X	100.0
	Y	87.5
4	X	100.0
	Y	75.0

The results presented in Table 5.3 require an understanding of the specific nuances of each task and respective type of problem in order to be interpreted correctly. In tasks of type (i.) the fix suggestion is presented immediately, and those who did not use the extension in tasks 1 and 2 either decided against using the suggestion or did not see it. In tasks of types (ii.) and (iii.), however, there might be no bug fix suggestion, as mentioned before. As a matter of fact, every participant

that was shown a suggestion in tasks 3 and 4 accepted it. The participant who did not accept a fix suggestion in task 3 of *Problem Set X* fixed the existing bug before filling in the missing code and thus was never provided with one. Similarly, the two participants who did not accept a suggestion in task 4 of *Problem Set Y* never introduced a bug in the first place (see problem type (iii.)), and thus the extension did not suggest a repair either.

5.2.3 Post-Test Survey

A post-test survey was completed by each participant after solving the eight problems. This section was split into three parts, each with a different objective. The first, consisting of statements *PT1* through *PT3*, aimed to understand whether the participants felt their performance was impacted in any way by the test environment. The second part, formed by the five statements from *PT4* through *PT8*, is used to assess user satisfaction while using the extension. Finally, the last three statements, *PT9* through *PT11* make up a section on future improvements, as a way to estimate what users would require from a tool of this kind. The post-test survey was as follows:

PT1: The second part was easier because I was already used to the code editor and the environment.

PT2: The development environment used (OS, IDE, extensions, etc.) is similar to the one I'm used to.

PT3: I would have been faster if I was using my own development environment.

PT4: Overall, the extension was helpful in fixing these bugs.

PT5: Overall, the suggested fixes were acceptable solutions.

PT6: I found it easy to understand the proposed solutions.

PT7: The live (real-time) aspect of the extension allowed me to be faster in fixing the bugs in question.

PT8: I would use an extension like this for my daily work.

PT9: An extension like this would be more helpful for more complicated bugs than for simple bugs.

PT10: The extension needs to be faster in providing solutions.

PT11: Pointing out the bug location is as helpful as providing a solution.

The frequency of responses to the first three statements suggests that the test environment did not affect the results of the experiment, as *PT2* shows a notorious similarity between the environment of the test and that each participant was used to (see Figure 5.5) and *PT3* an overall feeling that using their own development environment would not provide significant performance advantages. *PT1*, while divisive, seems to indicate that the order in which the tasks were performed did not introduce a bias in the second problem set due to familiarity with the environment from the first one.

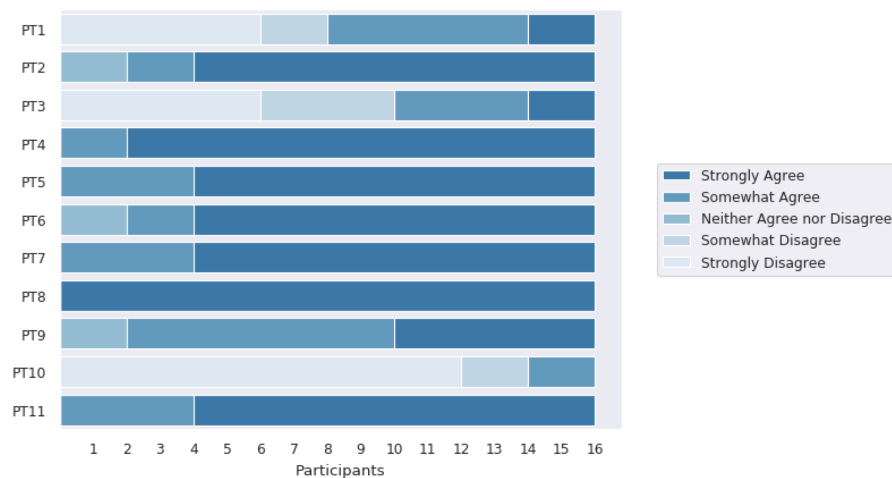

Figure 5.5: Stacked Bar Chart with frequencies of each *Post-Test* survey item.

Regarding the overall satisfaction of each participant, answers were overwhelmingly in agreement with the usefulness of the extension. Statements *PT4* and *PT5* show a general acceptance of the suggested fixes, with them being deemed acceptable solutions and the extension helpful, while *PT6* relates to how difficult is the understanding of each suggestion, which the participants consider easy to do. Perhaps most importantly for the problem here studied, *PT7* suggests that the *liveness* characteristic of this particular system was considered useful by the study participants. Lastly, every response to *PT8* was *Strongly Agree*, as the usefulness an extension of this kind was unanimous.

The final three questions, while not directly related to this particular extension, are still useful to draw conclusions over what worked in this experiment and what deserves further study. Participants largely considered that a live Automated Program Repair tool such as this extension would be more useful for more complicated bugs, as evidenced by the responses to *PT9*. Conversely, the consensus was that the current speed of the extension is not a limiting factor, as *PT10* show major disagreement with the statement that the extension needs to be faster in providing solutions. Finally, in *PT11* participants agree that pointing out the bug location is as helpful as providing a solution.

5.2.4 Discussion

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

H_1: Using a live Automated Program Repair tool improves the speed and final result of code solutions.

With it, we expected to be able to answer the following Research Questions:

- ***RQ1:*** *Are users faster in reaching the solution when using a live Automatic Program Repair tool?*

  Regarding **RQ1**, we compared the times that users took to solve each problem — described in detail in Section 5.2.2.1. In doing so, we can reject the null hypothesis that there are

no statistical differences for all tasks of type (i.), and half of those of types (ii.) and (iii.). Interpreting these results requires a deeper look at what each specific problem entailed and the significance of these results. As Task 1 and 2 were simple sanity checks, we now focus our discussion on Tasks 3 and 4. Concerning Task 3 (Problem Set Y), for which we were unable to reject the null hypothesis, it seems that most participants were not familiar with the reduce function, and this fact might have led considerably slower times to reach a solution; this, however, does not appear to be a direct threat to our main hypotheses and thus should provide us guidance for a better design of the validation package in the future. Similarly, both X and Y variants of Task 4 are extremely similar, which might introduce a bias in the participants that were already exposed to one of them.

Nonetheless, (1) we can still observe statistically relevant differences between the groups in 75% of the tasks, and (2) the mean time to reach a solution is consistently lower for the groups using the extension. Ergo, we argue that there exists substantial evidence to convince that **users are faster to reach a solution** when using a live Automatic Program Repair tool.

- ***RQ2:*** *Are solutions generated by an Automatic Program Repair tool different from the ones developed by human programmers?*

  The answer to **RQ2** lies in the situation described in Section 5.2.2.2, as solutions generated by the Automated Program Repair tool **are sometimes different** from those developed by human programmers. Although we are not able to understand if the generated code is better or worse than human-written code, it's possible to connect this to the problems discussed in Section 2.2.3 of overfitting and lack of maintainability of automatically generated patches. We do not make any formal test on whether the solutions generated by the tool developed as part of this dissertation share these problems, but we have identified situations in which the solution appears to be worse at least in terms of closeness to its meaning in natural language.

- ***RQ3:*** *Are users aware of the rationale of solutions generated by the Automatic Program Repair tool before accepting them?*

  The final Research Question, **RQ3**, is more complex. In fact, if we go by the results of the survey (see *X8* and *Y8* in Section 5.2.2.3), we'll see that there was no clear tendency to agree nor disagree with having tried to understand the solutions before accepting them. In *Problem Set Y* participants were equally split, while in *Problem Set X* there was a slight skew towards agreeing, but in doing a critical analysis of these responses, one must remain skeptical. Based on assorted notes taken during each test, we do not believe the majority of participants even tried to understand the problem, much less the proposed repair, in situations in which the suggestion was immediate. The extremely low mean times to reach a solution, such as *0:22* in task 1 of *Problem Set Y* (see Table 5.2), seem to support that observation, as we believe it's possible to argue that 22 seconds (or less) is not enough time to read through the problem, read the faulty code and understand the suggested repair. Considering the very slight tendency displayed in *X8* and *Y8* and the aforementioned observations, we believe the

experimental study was **not conclusive** as to whether participants tried to understand the suggested repairs or not.

Overall, our research points towards a live Automated Program Repair tool improving the speed of developers when presented with faulty programs, thus validating that part of Hypothesis $\mathbf{H_1}$, but does not provide enough evidence to either support or reject the notion that it also improves the final result of code solutions.

5.3 Threats to Validity

As with any empirical study, a number of threats to validity must be considered and the respective possible impact studied [PPV00]. In this section, threats to *Construct Validity*, *Internal Validity* and *External Validity* are described and a brief discussion on if and how each of them was addressed ensues.

5.3.1 Construct Validity

The first type of validity we consider is *Construct Validity*, which pertains to how well the tasks, environment, data collected and other variables in an empirical study map the constructs or hypothesis they represent [SCC02].

Validity of Problem Sets.

Likely the most important threat to construct validity in this study is the degree of fidelity to which the problem sets used in the experiment represent faulty programs that a developer is likely to find in his daily work. An attempt to do so at a high degree consisted in defining three different types of problems and in using existing code, rather than code written for the sole purpose of being used in this experiment (see Section 5.1.2).

Hypothesis Guessing.

Being under a test setting, some participants might try to guess the hypothesis and try to perform *better* based on that [WRH+12]. Quoting Marilyn Strathern's formulation of Goodhart's law, *"When a measure becomes a target, it ceases to be a good measure"* [Str97], and thus a variable that is supposed to model a specific construct or hypothesis might not do so as accurately if it starts being a target in itself. While it's not possible to be certain that a person is trying to aim for a specific result, we opted to conceal the hypothesis and objectives of the study and the measures we were taking until the end, in an effort to minimize the effect of this threat.

5.3.2 Internal Validity

The extent to which a valid *cause-effect* relationship can be established between the independent variables and the effects observed in the dependent variables is defined as *Internal Validity* [WRH+12].

Motivation of Participants.

In our study, we tried to analyze the form of the code and whether the participants attempted to understand the suggestions provided by the extension, revealing that participants often accepted suggestions provided by the extension without trying to understand them first, leading to, at least, suboptimal solutions. It is not clear whether these finding would stand if the faults they were repairing was in code they were going to maintain in the future or code that was going to be used in a production environment. It's perfectly reasonable to hypothesize that being in a test environment leads to a more relaxed attitude as there are no *real* stakes. Future experiments should take this threat into consideration, accounting for the motivation of participants.

Development Environment.

Software Developers are often very careful in setting up their development environment, with discussion on which Operating System, Text Editor or Unix shell is *the best* and therefore it's easy to see how a developer's performance might vary with a change of environment. We attempted to understand if this was an issue in our experiment by asking three different questions related to it in the Post-Test survey (see Section 5.2.3). Responses show that not only was the development environment similar to the one used by the vast majority of the participants, but they also don't believe using their own environment would have provided any significant performance benefits.

Physical Environment.

Each participant completed the test in the Software Engineering laboratory at the Faculty of Engineering of the University of Porto, but the room was not sealed for the purposes of this experiment. Owing to that, noise or other distracting factors, which were not constant for all participants, could have had an impact on the test results. In order to try and reduce this impact, we attempted to perform the tests at times of low entropy, but future experiments should try to control these variables in a more rigorous manner.

5.3.3 External Validity

The final type of validity discussed in this section is *External Validity*, which concerns the degree to which the findings of a study can be generalized to different populations or settings [WRH+12].

Sample Size.

In the empirical experiment here described there were 16 participants, which were split into two groups eight members each. This is a rather small sample size, something that must be taken into account when interpreting the findings of this study.

Sample Characteristics.

Besides the size of the sample, its composition is also of importance to the validity of the study. In particular, all but one of the participants are students in the final year of

Informatics and Computing Engineering at the Faculty of Engineering of the University of Porto. Consequently, our sample displays a level of homogeneity that does not correspond to that of the population of Software Developers in general. However, findings from studies such as the one from *Salman, Misirli and Juristo* show students do not perform better nor worse than software engineering professionals when using new technologies during experimentation [SMJ15], as is the case here. Regardless, future experiments should attempt to introduce a higher degree of heterogeneity in terms of participant background.

5.4 Summary

In this chapter, we presented a thorough account of the experimental study. In Section 5.1 we described the methodology to follow, including the overall plan and the specific tasks the study would be made up of.

Then, in Section 5.2, we presented the results obtained in the experiment, accompanied by a descriptive analysis and an explanation for the methods used, followed by a critical discussion of these results and a interpretation of the findings in relation to the hypothesis posed and research question under study.

In Section 5.3, we presented the threats to the validity of the study, while mentioning steps taken to reduce their impact or suggestions for future studies.

Chapter 6

Conclusions

6.1 Conclusions 51
6.2 Main Contributions 52
6.3 Future Work 53

This chapter reflects on this document and the work here presented. Firstly, Section 6.1 we present the conclusions of this dissertation. We then summarise, in Section 6.2, the main contributions resulting from this work. Lastly, we discuss future work regarding live Automated Program Repair tools in Section 6.3.

6.1 Conclusions

This work started with an exploration of the state of the art in Automated Program Repair and current challenges. In addition, we discussed one of the limitations of current code completion tools, which provide only *syntactic* suggestions. Combining these two areas, we proposed that a live Automated Program Repair tool could offer *semantic* suggestions to developers while coding. Our hypothesis was as follows:

H_1: Using a live Automated Program Repair tool improves the speed and final result of code solutions.

In order to validate the proposed hypothesis, we developed a prototype of one such tool, implemented as an extension to *Visual Studio Code*, repairing *JavaScript* code and offering suggestions to the developers in real-time, using a mutation-based approach to Automated Program Repair in order to generate those patches. Then, we performed an empirical study, in which we evaluated the performance of software developers in repairing bugs, both while using our tool and without it.

These experiment was used to validate the aforementioned hypothesis and answer the following Research Questions:

- ***RQ1:*** *Are users faster in reaching the solution when using a live Automatic Program Repair tool?*

  We argue that there is evidence to support the claim that **users are faster to reach a solution** when using a live Automatic Program Repair tool, as statistically relevant differences between the group using the extension and the other are found in 75% of the tasks and the mean time to reach a solution is lower for all of the tasks when using the implemented prototype.

- ***RQ2:*** *Are solutions generated by an Automatic Program Repair tool different from the ones developed by human programmers?*

  While we make no claims as to whether the generated solutions are better or worse than human-written ones, we did find evidence of them being **sometimes different**. In addition, we argue, based on direct observation during the experimental study, that some solutions generated by the extension are not as close to their meaning in natural language as the human-written solution are.

- ***RQ3:*** *Are users aware of the rationale of solutions generated by the Automatic Program Repair tool before accepting them?*

  Our findings were **not conclusive** regarding this Research Question, as the responses to the survey were mostly split and the very slight tendency they did show was directly contradicted by our own observations.

Ultimately, we are able to validate at least part of Hypothesis $\mathbf{H_1}$, as we do find that the speed of developers in fixing faulty code improves while using a live Automated Program Repair Tool, but do not reach a conclusion regarding the improvement of the resulting code.

6.2 Main Contributions

The main contributions of this dissertation can be summarised as follows:

- A Visual Studio Code Extension implementing a **live Automated Program Repair tool**, which leverages unit tests and mutation-based Automated Program Repair techniques to generate patches to faulty code and suggests them to the developers in real-time, achieving a level of 5 in the Liveness scale, providing *tactically predictive* feedback;

- An **Empirical Study** with 16 participants using the extension developed during this work and assessing its usefulness. The results from this study are encouraging in terms of the possibilities in the space of live Automated Program Repair.

6.3 Future Work

Multiple improvements to the developed tool can be implemented, and different experiments can, and should, be conducted. Here we briefly summarise our thoughts for the future of this area of research:

- Implementing a tool with the same characteristics as the one resulting from this work, but using a different approach to program repair. This could involve introducing fault localization techniques, as well as delving into different and potentially more complex automated program repair strategies. The challenge here is maximizing the potential of the tool in terms of how many bugs it is capable to fix while minimizing the time in which it does so, in order to maintain its liveness factor;
- Offering more than one suggestion to the developer, when possible, providing him with a choice between different patches;
- Conducting other experiments, possibly based on the recommendations given in Section 5.3, and doing so with multiple different tools, in order to further explore the possibilities of live Automated Program Repair tools in general and not of a specific implementation.

References

[AB14] Fatmah Yousef Assiri and James 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. IEEE, 2014.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, January 2004.

[ARC+19] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development — tightening the feedback loops. In *Proceedings of the 5th Programming Experience (PX) Workshop*, apr 2019.

[Bal85] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Trans. Softw. Eng.*, 11(11):1257–1268, November 1985.

[BB12] Harry N Boone and Deborah A Boone. Analyzing likert data. *Journal of extension*, 50(2):1–5, 2012.

[BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[Bro11] James Dean Brown. Likert items and scales of measurement. *Statistics*, 15(1):10–14, 2011.

[BS99] Patrick D Bridge and Shlomo S Sawilowsky. Increasing physicians' awareness of the impact of statistics on research outcomes: comparative power of the t-test and wilcoxon rank-sum test in small samples applied research. *Journal of clinical epidemiology*, 52(3):229–235, 1999.

[CD94] Dennis L Clason and Thomas J Dormody. Analyzing data measured by individual likert-type items. *Journal of agricultural education*, 35(4):4, 1994.

[CDE+08] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on software engineering*, (3):215–222, 1976.

[CP08] James Carifio and Rocco Perla. Resolving the 50-year debate around using and misusing likert scales. *Medical education*, 42(12):1150–1152, 2008.

[CTBB11] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 121–130. ACM, 2011.

[DC02] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130, 2002.

[DCJ06] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with sat. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 109–120. ACM, 2006.

[Dij89] Edsger W Dijkstra. On the cruelty of really teaching computing science. *Communications of the ACM*, 32(12):1398–1404, 1989.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[DW10] Vidroha Debroy and W Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.

[DW13] Joost CF De Winter. Using the student's t-test with extremely small sample sizes. *Practical Assessment, Research & Evaluation*, 18(10), 2013.

[FDN14] Alfonso Fuggetta and Elisabetta Di Nitto. Software process. In *Proceedings of the on Future of Software Engineering*, FOSE 2014, pages 1–12, New York, NY, USA, 2014. ACM.

[FLW12] Zachary P Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2012.

[FNWLG09] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 947–954. ACM, 2009.

[FPG94] Norman Fenton, Shari Lawrence Pfleeger, and Robert L. Glass. Science and substance: A challenge to software engineers. *IEEE software*, 11(4):86–95, 1994.

[GMK11] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using sat. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 173–188. Springer, 2011.

[GZ13] Vahid Garousi and Junji Zhi. A survey of software testing practices in canada. *Journal of Systems and Software*, 86(5):1354–1376, 2013.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[HG04] Haifeng He and Neelam Gupta. Automated debugging using path-based weakest preconditions. In *International Conference on Fundamental Approaches to Software Engineering*, pages 267–280. Springer, 2004.

[J+04] Susan Jamieson et al. Likert scales: how to (ab) use them. *Medical education*, 38(12):1217–1218, 2004.

[JH05] 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*, pages 273–282. ACM, 2005.

[JJE14] 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*, pages 437–440. ACM, 2014.

[KB11] Robert Könighofer and Roderick Bloem. Automated error localization and correction for imperative programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, pages 91–100. FMCAD Inc, 2011.

[KB12] Robert Könighofer and Roderick Bloem. Repair with on-the-fly program analysis. In *Haifa Verification Conference*, pages 56–71. Springer, 2012.

[KHL11] Damir Kalpić, Nikica Hlupić, and Miodrag Lovrić. *Student's t-Tests*, pages 1559–1563. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[Koz94] John R Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and computing*, 4(2):87–112, 1994.

[LB12] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *ACM SIGPLAN Notices*, volume 47, pages 133–146. ACM, 2012.

[LBF+12] Francesco Logozzo, Michael Barnett, Manuel A Fähndrich, Patrick Cousot, and Radhia Cousot. A semantic integrated development environment. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 15–16. ACM, 2012.

[Lev61] Howard Levene. Robust tests for equality of variances. *Contributions to probability and statistics. Essays in honor of Harold Hotelling*, pages 279–292, 1961.

[LGDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.

[LGFW13] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software quality journal*, 21(3):421–443, 2013.

[LGNFW12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *Ieee transactions on software engineering*, 38(1):54, 2012.

[Lik32] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[LR15] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.

[LR16] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices*, 51(1):298–312, 2016.

[Mey92] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[Mon18] Martin Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, January 2018.

[MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th international conference on software engineering*, pages 691–701. ACM, 2016.

[Nor10] Geoff Norman. Likert scales, levels of measurement and the "laws" of statistics. *Advances in Health Sciences Education*, 15(5):625–632, Dec 2010.

[NQRC13] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: Program repair via semantic analysis. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 772–781. IEEE, 2013.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015.

[OVH+17] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A Hammer. Toward semantic foundations for program editors. *arXiv preprint arXiv:1703.08694*, 2017.

[PFN+14] Yu Pei, Carlo A Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. Automated fixing of programs with contracts. *arXiv preprint arXiv:1403.1117*, 2014.

[PFNM15] Yu Pei, Carlo A Furia, Martin Nordio, and Bertrand Meyer. Automated program repair in an integrated development environment. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 681–684. IEEE Press, 2015.

[PHH+18] Justyna Petke, Saemundur O Haraldsson, Mark Harman, William B Langdon, David R White, and John R Woodward. Genetic improvement of software: a comprehensive survey. *IEEE Transactions on Evolutionary Computation*, 22(3):415–432, 2018.

[PPV00] 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*, pages 345–355. ACM, 2000.

[QML+14] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265. ACM, 2014.

[RFS+17] Simone Romano, Davide Fucci, Giuseppe Scanniello, Burak Turhan, and Natalia Juristo. Findings from a multi-method study on test-driven development. *Information and Software Technology*, 89:64–77, 2017.

[RG16] Bat-Chen Rothenberg and Orna Grumberg. Sound and complete mutation-based program repair. In *International Symposium on Formal Methods*, pages 593–611. Springer, 2016.

[RT96] C. V. Ramamoorthy and Wei-Tek Tsai. Advances in software engineering. *Computer*, 29(10):47–58, October 1996.

[RW88] Charles Rich and Richard C. Waters. Automatic Programming: Myths and Prospects. *Computer*, 21(8):40–51, 1988.

[SBLGB15] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[SCC02] William R Shadish, Thomas D Cook, and Donald T Campbell. Experimental and quasi-experimental designs for generalized causal inference. 2002.

[SDM+18] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *Proceedings of SANER*, 2018.

[SMJ15] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 666–676. IEEE, 2015.

[Str97] Marilyn Strathern. ‘improving ratings’: audit in the british university system. *European review*, 5(3):305–321, 1997.

[SW65] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[Tan90] Steven L Tanimoto. Viva: A visual language for image processing. *J. Vis. Lang. Comput.*, 1(2):127–139, 1990.

[Tan13] Steven L Tanimoto. A perspective on the evolution of live programming. In *Proceedings of the 1st International Workshop on Live Programming*, pages 31–34. IEEE Press, 2013.

[TLPH95] Walter F Tichy, Paul Lukowicz, Lutz Prechelt, and Ernst A Heinz. Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, 28(1):9–18, 1995.

[WNLGF09] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374. IEEE Computer Society, 2009.

[WPF+10] 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*, pages 61–72. ACM, 2010.

[WRH+12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[WTL16] Fern K Willits, Gene L Theodori, and AE Luloff. Another look at likert scales. *Journal of Rural Social Sciences*, 31(3):126, 2016.

[YZLT17] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 831–841. ACM, 2017.