

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Property Tests as Specifications Towards Better Code Completion

Afonso Jorge Ramos



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, PhD

Second Supervisor: André Restivo, PhD

July 30, 2020

Property Tests as Specifications Towards Better Code Completion

Afonso Jorge Ramos

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: João Pascoal Faria

External Examiner: João Miguel Fernandes

Supervisor: Hugo Sereno Ferreira

July 30, 2020

Abstract

Despite significant advancements in modern software development, size, complexity, and intractability are also growing rampantly, leading to an increased focus on the software development process and requiring shorter bug fixing and maintenance cycles, to simplify and solidify the final product. One of the barriers toward this shift has been the time consuming, painful, and expensive process of debugging. Recent studies have shown that around 50% of the development process comes from debugging, from the exhaustive effort to identify failed executions, the process of implementing the fixes, to the validation of said fixes, which are mainly executed manually, all contribute to this problem.

Automatic Programming is a research field that has seen a growing interest in latest years. This field proposes the synthesis of programs from requirement specifications or higher-level abstractions, automatizing the development process using a set of different techniques. Automatic Program Repair (APR) focuses on the identification, location, and production of bug-fixes, alleviating the developer from such efforts and thus phase out the daunting debugging time from the development cycle. Nevertheless, a myriad of APR software solutions exist, many have been around for a long time, which can be achieved with a wide range of techniques, algorithms, and heuristics, with fixes typically arising from the existence of a well-formed test-suite.

Current APR solutions' integration with an IDE is almost non-existent, and, as such, we propose a code completion tool capable of providing semantic suggestions, in addition to the standard IDE syntactic suggestions, as tactically predictive live feedback. Most APR solutions use test suites as specifications, though, most of these are in the form of example-based testing, requiring numerous test cases to check for all potential problems, which becomes undesirably burdensome and error-prone. In this work, we argue that the use of Property-Based Testing (PBT), a testing technique that generates random input data to verify the expected behaviour, is an improved specification to form semantic code completion suggestions providing developers with an improved live automatic program repair tool without risking test overfitting.

Our implemented solution, unifies these concepts, and brings to developers an open-source tool, that places itself ahead of the developer, requiring a not as sizeable quality of suggestions, by merely improving developers' reasoning with the added bug localisation and fix suggestions, through mutation generation. Herewith, our solution, achieves a shorter development cycle, better code quality and, facilitates coding by accelerating the repair time, which can be observed in our results. Furthermore, in our results, we observed a wide acceptance of PBT frameworks, as well as a treacherous trust in incomplete example-based test suites, proving the need for PBT in Test Driven Development (TDD). Finally, an overall acceptance and trust of the tool was obtained, accompanied by a shrinking scepticism of APR through the use of our tool.

Keywords: Live Automatic Programming Repair (LAPR), Automatic Programming Repair (APR), Property-Based Testing (PBT), Test Driven Development (TDD)

ACM Categories: Software and its engineering \Rightarrow Software creation and management \Rightarrow Software verification and validation \Rightarrow Software defect analysis \Rightarrow Software testing and debugging;
Software and its engineering \Rightarrow Software creation and management \Rightarrow Software development techniques \Rightarrow Automatic programming

Resumo

Apesar dos avanços significativos no desenvolvimento de software moderno, o tamanho, a complexidade e a intratabilidade do software cresce de forma igualmente acelerada, focando-se, cada vez mais, no processo de desenvolvimento de software e reduzindo os ciclos de correção e manutenção de erros, para simplificar e solidificar o produto final. Uma das barreiras para essa mudança foi o processo demorado, doloroso e caro de depuração. Estudos recentes demonstraram que cerca de 50 % do processo de desenvolvimento é originado na depuração de código, da procura por falhas de implementação das correções e das validação dessas mesmas correções, que, nos dias de hoje, são ainda executadas manualment.

A programação automática, um campo que tem vindo a despoletar um crescente interesse, propõe a síntese de programas a partir de especificações de requisitos ou abstrações de nível superior, automatizando o processo de desenvolvimento usando um conjunto de técnicas diferentes, tem aumentado a níveis nunca antes vistos. O conceito de *Automatic Program Repair* (APR) concentra-se na identificação, localização e geração de correções de bugs, aliviando o programador de tais esforços, e, dessa forma, eliminando o assustador tempo de depuração dos processos de desenvolvimento. No entanto, existem já inúmeras soluções de APR, muitas destas há muito tempo, que alcançam os seus objetivos com uma ampla gama de técnicas, algoritmos e heurísticas, com correções normalmente decorrentes da existência de um conjunto de testes bem formado.

Soluções atuais de APR possuem, no entanto, uma integração em IDEs quase inexistente e, como tal, aqui propomos uma ferramenta de conclusão de código capaz de fornecer sugestões semânticas, além das sugestões sintáticas padrão de IDEs, como *live feedback* taticamente preditivo. A maioria das soluções APR usa suítes de testes como especificações, no entanto, a maioria destas é na forma de testes baseados em exemplos, exigindo vários casos de teste para verificar todos os potenciais problemas, algo que facilmente se torna indesejável, dispendioso e propenso a erros. Neste trabalho, argumentamos que o uso de testes baseados em propriedades (PBT), uma técnica de teste que gera dados de entrada aleatórios para verificar o comportamento esperado, é uma especificação aprimorada para formar sugestões semântico de *code completion*, fornecendo aos desenvolvedores uma solução de live APR.

A solução implementada unifica estes conceitos, fornecendo aos desenvolvedores uma ferramenta de código-fonte aberto, que se coloca à frente do mesmo, o que exige uma qualidade não tão alta de sugestões, simplesmente melhorando o raciocínio dos desenvolvedores com a localização de bugs e sugestões de correção, através de geração de mutantes. Assim, a nossa solução alcança um ciclo de desenvolvimento mais curto, melhorando a qualidade do código, facilitando a programação e acelerando o tempo de reparo, o que pode ser observado nos nossos resultados. Além disso, nos nossos resultados podemos observar uma crescente aceitação de testes baseados em propriedades, bem como uma confiança, por parte dos participantes, em conjuntos de testes incompletos baseados em exemplos, provando a necessidade de PBT no desenvolvimento orientado a testes. Finalmente, foi obtida uma aceitação e confiança gerais da ferramenta, acompanhadas de um ceticismo cada vez menor de soluções APR pelo uso de nossa ferramenta.

Acknowledgements

First and foremost, my sincere gratitude goes to my parents, who have always supported me in all my life choices, provided me with all the tools to succeed, and guided me through life, while continuously staying the greatest example of who to be. They, alongside with my sister, have always been there for me, both in physical presence, which was increased in the past few months, and in emotional presence, for their continued support, constant worry and help with my unfortunate difficulties. I cannot forget to mention Zoey, who has been a great companion through a lot of days during this pandemic where my happiness was hard to grasp, but she always found a way of helping me let go of my troubles.

To the friendships that feel formed decades ago, the friendships formed in the start of this *feupian* journey (a spoonful of *HUMUS* is always good, especially those taken during all-nighters in Discord), and the friendships that have only strengthened in the past two years, an enormous thank you, because with you I have always felt happy, and you have helped me grow in ways I could never imagine.

At last but not least, an extreme thank you to André and Hugo, who always motivated me to do better, pushing for improvements, and always expecting the best of me. Their patience, guidance and support were crucial to the proud conclusion of this dissertation.

Afonso Ramos

*“Talk is cheap.
Show me the code.”*

Linus Torvalds

Contents

1	Introduction	1
1.1	Context	1
1.2	Problem Definition	2
1.3	Motivation	3
1.4	Objectives	4
1.5	Document Structure	4
2	State of the Art	7
2.1	Software Engineering	7
2.2	Software Development Process	8
2.3	Live Programming	9
2.4	Automatic Programming	10
2.5	Automatic Program Repair	11
2.5.1	Generate and Validate Program Repair	14
2.5.2	Semantics-Based Program Repair	16
2.6	Property-Based Testing	17
3	Problem Statement	19
3.1	Overview	19
3.2	Issues	20
3.3	Hypothesis and Research Questions	22
3.4	Validation Methodology	23
3.5	Proposal	23
4	Proposed Solution	25
4.1	Contextualization	25
4.2	Objectives	25
4.3	Implementation	27
4.3.1	Automated Program Repair	28
4.3.2	Language Server Protocol	29
4.3.3	Mutation Generation	31
4.3.4	pAPRika Extension	37
4.3.5	Property-Based Testing Framework	41
4.3.6	TypeScript Support	42
4.3.7	General Improvements	42
4.4	Summary	43

5	Empirical Evaluation	45
5.1	Objectives	45
5.2	Guidelines	46
5.3	Planning	47
5.3.1	General Public Questionnaire	47
5.3.2	Usability Questionnaire	48
5.4	Tasks	50
5.4.1	General Public Tasks	50
5.4.2	Usability Tasks	52
5.5	Results	55
5.5.1	Participants' Profile	56
5.5.2	Background	56
5.5.3	Post-Test Survey	57
5.5.4	Usability Questionnaire	58
5.5.5	General Public Questionnaire	63
5.6	Threats to Validity	69
5.6.1	Construct Validity	69
5.6.2	Internal Validity	70
5.6.3	External Validity	70
5.7	Discussion	71
5.8	Summary	75
6	Conclusions	77
6.1	Summary	77
6.2	Main Contributions	78
6.3	Future Work	79
A	General Public Questionnaire	81
B	Usability Questionnaire	89
	References	101

List of Figures

2.1	Software Development Life Cycle example.	9
2.2	Extended version of the liveness hierarchy	10
2.3	Timeline of the evolution of the research on test oracles	12
2.4	Generate and Validate repair process	15
2.5	Semantics-driven repair process	16
4.1	Flowchart of the automated program repair process	27
4.2	Difference between an implementation without LSP versus one with it.	29
4.3	Example notification exchange between pAPRika client and server.	30
4.4	Tool's settings within the settings menu of Visual Studio Code.	38
4.5	Display of the tool's progress on the bottom left corner of Visual Studio Code. . .	38
4.6	Underlined potential fix locations by pAPRika.	39
4.7	Underlined potential fix location in Diogo Campos' tool	40
4.8	List of problems generated by the tool.	40
4.9	Deployment of fix suggestions.	41
5.1	Sample of example-Based tests for <code>myParseInt</code>	51
5.2	Sample of example-Based tests for <code>longestCommonSubstring</code>	51
5.3	Example properties for <code>myParseInt</code>	52
5.4	Example properties for <code>longestCommonSubstring</code>	52
5.5	Sample of example-Based tests for <code>myParseInt</code>	53
5.6	Sample of example-Based tests for <code>longestCommonSubstring</code>	53
5.7	Bar chart with the participants' most important features of the tool.	63
5.8	Bar chart with the participants' features of the tool that should be improved on. .	63
5.9	Pie chart with the highest completed degree of education of participants.	64
5.10	Histogram with the number of participants per years of professional experience. .	64
5.11	Bar chart for task 1's <code>myParseInt</code> assessment.	65
5.12	Bar chart for task 1's <code>longestCommonSubstring</code> assessment.	65
5.13	Bar chart for task 2's <code>myParseInt</code> assessment.	66
5.14	Bar chart for task 2's <code>longestCommonSubstring</code> assessment.	66
5.15	Bar chart for task 3's <code>myParseInt</code> assessment.	67
5.16	Bar chart for task 3's <code>longestCommonSubstring</code> assessment	68
5.17	Bar chart with the participants' most important features of the tool.	69
5.18	Answers to the post-test section of both questionnaires.	73

List of Tables

5.1	Statistical measures and p-value for hypothesis tests on background scores. . . .	58
5.2	Statistical measures and Mann–Whitney U (p) of time to solve each problem set.	59
5.3	Scoring of questions per problem, according to the use of the tool.	61
5.4	Comfortability score, per participant, for questions F1 to F6, based on the use of this tool.	62
5.5	Statistical measures of background scores.	64
5.6	Comfortability score, per participant, for questions F1 to F6, based on the provided gifs.	68
5.7	Levene test and Student t-test, for questions F1 to F6, between both questionnaires.	72

Abbreviations

APR	Automatic Program Repair
AST	Abstract Syntax Tree
CI	Continuous Integration
CD	Continuous Deployment
IDE	Integrated Development Environment
LHS	Left Hand Side
LSP	Language Server Protocol
PBT	Property-Based Testing
RHS	Right Hand Side
TDD	Test-Driven Development
VS	Visual Studio

Chapter 1

Introduction

1.1	Context	1
1.2	Problem Definition	2
1.3	Motivation	3
1.4	Objectives	4
1.5	Document Structure	4

This chapter establishes the subject, motivation, and scope of this dissertation and, then, proceeds to define the problem. Firstly, Section 1.1 provides an overview of the context surrounding this work, followed by Section 1.2 which identifies the problem meant to be solved. Section 1.3 clarifies the importance of these areas of work and why it deserves the attention it has been having, and Section 1.4 describes the objectives of this dissertation. Finally, the structure of this document and its content is outlined within Section 1.5.

1.1 Context

In a world full of instant feedback and knowledge at the distance of a click, software development has also delved into the immediacy of instructions and/or suggestions. Negative effects of such minute interaction of social media aside, for most intellectual activities, this instantaneous feedback has been proven superior to delayed feedback, even increasing the gain of expertise [KKH13, KA72, EKS93]. Whence modern software development has seen growing advancements in programming environments that have allowed similar reactive feedback when software features and implementations change. These aforementioned advancements are, somewhat, a result of the ever-growing size, complexity, and intractability of software, which reflects the critical resource that it is within modern society, posing unique and difficult challenges, and stress on software development organizations and teams [FD14]. And it is well known that great responsibility is the inseparable consequence of great power [Fra93], therefore we must ensure the trust and reliability

of software and, one way to do so is to increase mastery of software developers and reduce risks of failure.

Accordingly, providing developers with tools that provide valuable real-time feedback serves to mitigate manual bug fixing, reduce time and cost spent for debugging, which is estimated to reach up to 50% of the development process [BJ07], and to automate the software development process. Furthermore, Software is one of the most crucial and complex components of any sophisticated product or service, and as Lehman’s laws of software evolution state, when functionality grows, there is a matching decrease in quality and increase in complexity [Leh80, LR01], as such, for quality to be guaranteed, additional methodologies must be found. To reach this goal, many frameworks have been created, however, many have two big elephants in the room, the lack of liveness, as per S. Tanimoto [Tan13], and the fact that most patches are overly specific to the specifications by exploiting their specificities and weaknesses [SBLB15]. As per the latter, fixes fail to generalise to the held-out tests, ending up, borrowing the term from machine learning, overfitted. Given this problem, D. Campos [Cam19] created such a framework revamped the feedback loop of the edit-compile-run software development cycle towards a continuum, *i.e.*, accelerating and facilitating coding for the developer. However, such a solution clears only the smallest of the two elephants, since it does not fix the overfitting created by the necessity of extensive unit tests as specifications. Subsequently, we hypothesise that we can rectify this problem by using property-based testing as a specification, further expanding the tested area, augmenting the probability of finding bugs.

Reframing, Software must not be seen as a commodity, as it has a cost associated, especially when, nowadays, most of the development costs of software systems befall post-deployment through maintenance and evolution [MS06] and, consequently, reducing the total development time of a software system is highly beneficial for society as a whole. From the myriad of techniques and approaches, in this work, we will be focusing on automating the steps of the developing and debugging process, by providing automatic build tools and intelligent semantic code completion from property-based unit tests.

1.2 Problem Definition

The rising complexity of software systems has created a very high interest in tools that allow developers to create, design and debug robust and safe software faster while preserving or even increasing its quality, robustness, and maintainability. Thus, current Automated Program Repair (APR) solutions are already present and used within the scientific community which approach or delve into this fundamental obstacle, both with studies and tools. However, such tools are still awfully underused within software development teams and organizations due to several obstacles that prevent them from their wide use. Therebetween, the necessity of specification correctness and completeness to properly repair automatically using current techniques and the sometimes unreadable, and of questionable maintainability, machine-generated repairs. To achieve specification completeness, its creator has to think of all the possible cases, though if such test cases can be

thought of, they are probably already well regarded during their development, ending up with success during the testing phase. Therefore, there is a need to reach the test cases that the test creator cannot reach. Additionally, a problem that almost no APR solution has yet been able to repair, is the matter of overfitting, as most patches are overly specific to specifications, failing to customize the fix to the problem instead of to the solution.

Simultaneously, code-completion, an integral part of modern Integrated Development Environments (IDEs), is highly useful to accelerate code-writing and to assist developers in avoiding typos. Conventional code completion systems are ad-hoc and neither complete nor sound and propose exclusively static type suggestions of the programming language, consequently, it is common for such suggestions to be irrelevant for a specific working circumstance. Nonetheless, the suggestions presented happen to stay a step ahead of the programmer and provide live feedback, *i.e.*, provide tactically predictive feedback [Tan13], even if solely static.

This dissertation seeks to research whether we can kill both birds with one stone, by taking advantage of APR solutions to suggest semantic suggestions to the developer while using properties to define specifications. As this solution needed not to repair software after its development, but rather during, it would introduce a human step into the process, taking advantage of both the machine suggestions and the developer's reasoning, as the repairs would be merely suggestions, mitigating the readability and maintainability concerns previously mentioned. Automated program repair is an emerging and thrilling field of research that enables automated rectification of bugs and vulnerabilities in software, and by presenting such a panacea, we intend to present a modern twist to the automated program repair field.

1.3 Motivation

Since software still cannot be seen as a commodity, and most of the development costs of software systems befall post-deployment through maintenance and evolution [MS06] with some reports claiming that modifications made after its first delivery can reach up to 90% of the total project cost [SPL03], reducing such costs are of the utmost importance. The predicament of never-ending maintenance is mostly caused by none other than bugs, the colloquial term for programming mistakes, a task which can be time-consuming, difficult and tediously manual. Such plight has encouraged a wide range of work on their automatic identification and elimination. For this identification and elimination, *i.e.*, APR solutions, which constitutes a substantial part of software maintenance, countless uses can be found, from improving programmer productivity to outright generating hints for following with specification. Nevertheless, we aim to reduce the time and cost spent during both debugging and development, mitigate the cumbersome task of manual bug fixing and to automate the software development process.

Be that as it may, tools must integrate well with development environments to achieve wide adoption, and even if it is something that has been achieved by some research groups, none have reached the desired adoption. Regardless, the most prominent challenge in today's research on APR is weak specifications, which we intend to solve using Property-Based Testing (PBT), where rather

than writing a myriad of unit tests by hand, which in turn may cause test overfitting, only general properties must be specified. With such implementation and approach, we manage to create a simple to use, useful, and efficient framework that can achieve valuable real-time feedback empowering developers to build software faster by squeezing the edit-compile-run software development cycle into a pulp.

1.4 Objectives

Software's increasing size, complexity, and intractability are creating strong barriers to fast and efficient development. Furthermore, stricter release requirements are pressuring for shorter bug fixing and maintenance cycles, turning the responsibility to developers for timely developing high-quality software. For these reasons, the industry has seen an increase in the research of providing developer feedback as early on the development process as possible, evaluating if such changes influence developers, reduce their workload, and affect in what they produce positively.

Therefore, the purpose for this dissertation is threefold, that is, to gather the state of the art on the topics of automatic programming, unit testing, language server protocols, and respective tools; to create a selection of suitable approaches to automatically generate code completion suggestions based on existing specifications, in the form of properties; and, finally, to develop a plugin capable of suggesting smart semantic auto-completion, using live testing, to the developer, enabling both automated rectification of bugs and vulnerabilities, and accelerated development.

Culminating into our main research questions, which intend to gather evidence on whether it is possible to enhance the code's quality and development speed, by creating a framework capable of revamping the edit-compile-run software development cycle's feedback loop towards a continuum, effectively providing coding conveniences for the developer, as well as, harnessing the power of property-based testing.

1.5 Document Structure

The remainder of this dissertation is organised into the following five chapters:

- Chapter 2 (p. 7), **State of the Art**, introduces the background information and explanation about concepts necessary for the full understanding of this dissertation, as well as the state of the art of automatic program repair.
- Chapter 3 (p. 19), **Problem Statement**, lists the issues found in some solutions, presents the problems this dissertation aims to solve, as well as the approaches that are taken to solve it.
- Chapter 4 (p. 25), **Solution**, proceeds to provide an overview and description of the implemented solution in detail.
- Chapter 5 (p. 45), **Empirical Evaluation**, thoroughly describes the validation process and analyse its the answers to the research questions.

- Chapter 6 (p. 77), **Conclusions**, summarizes the work developed, concludes the dissertation, and includes a reflection on future contributions.

Chapter 2

State of the Art

2.1	Software Engineering	7
2.2	Software Development Process	8
2.3	Live Programming	9
2.4	Automatic Programming	10
2.5	Automatic Program Repair	11
2.6	Property-Based Testing	17

Throughout this chapter, we will briefly introduce, describe and contextualize most concepts surrounding the topic of this dissertation. Firstly, we introduce the very notion of Software Engineering (*cf.* Section 2.1), to then proceed to explain the increasing interest in the Software Development Process (*cf.* Section 2.2, p. 8), followed by the concept of live programming (*cf.* Section 2.3, p. 9). After these introductions, we delve into the distinct matters of automatic programming (*cf.* Section 2.4, p. 10), automatic program repair (*cf.* Section 2.5, p. 11), and property-based testing (*cf.* Section 2.6, p. 17).

2.1 Software Engineering

Software engineering is one of the many engineering curriculums, concerning software production phases from start to finish, from the forging of system specifications to system maintenance after deployment [Som10]. Software engineers, analogous to all engineers, have the task of making things work, applying methods, theories, and tools wherever necessary or possible, considering the whole development process, technical or not. Likewise, accepting that there are constraints to work within is also an important quality of any engineer, as it is intrinsic to achieve results of a certain quality standard without compromising on budget and schedule.

Since the first computer, the world has been growing exponentially on its dependence on software, with whole national infrastructures and utilities managed by code, thus, software engineering has become fundamental in the functioning of society. One of the problems *per se* of software

is that it is intangible and abstract, not constrained by physical laws, meaning that software can more easily get out of hand, getting harder to understand, complex and easily inflexible to changes. Ubiquity of software is also a problem, as the easiness with which one can develop software is increasing year by year without following any methods or techniques, which at first may seem a good thing, but it lowers reliability, expectations, and ends up increasing costs, as it can be seen in the development of Iowa's caucus app [War20] where the lack of adequate technical experience and high demands spurred chaos on an election night. In the end, software fails mostly due to the increasing demands of more complex systems without software engineering techniques on par with such needs, and low expectations from the increasing number of unqualified people working in the area.

All these concerns exacerbate the importance of software engineering, as society relies on it more and more with the need for achieving trustworthy and reliable systems while keeping costs and development speed low, however, costs are mostly post-deployment through maintenance and evolution [MS06]. In light of such obstacles, and as engineering is based on finding the best method for specific sets of situations, a systematic approach in software engineering is used, *i.e.*, software development process.

2.2 Software Development Process

With the ever-increasing relevance of software engineering, due to many modern products, processes, and services core elements' shifting towards software playing a central role, pressure accompanied by new and tough challenges are rising. Hence, software processes are created and specified to facilitate human comprehension, communication, and coordination; to describe, evaluate, automate, and improve procedures, techniques, and policies used for software development proficiency; and to assist with project management [BS14]. Like all intellectual and creative processes, software processes are complex and heavily rely on good judgment and decision-making. For these reasons, there has been a move towards standardization of said processes, to introduce new engineering methods and techniques, and good software engineering practices, but also to reduce diversity across and within organizations, therefore, reducing training required [Som10].

However, there are no perfect processes that work across all industries, therefore these change and adapt to improve the efficiency of the process itself and people following such process, the quality of the work products, and the development schedule [BS14]. The appearance and wide adoption of open source software, agile, and numerous cross-platform development frameworks have, though, shaken the software development and distribution practices, and now development teams are increasingly integrated within the product teams to minimize time to market. Nevertheless, these changes have brought some consequences to the software development life cycle, such as non-functional requirements related to performance and fault tolerance analysis shifting from the implementation to the development phase; systems moving towards becoming a combination of several development products and having a dynamic and on-the-fly reconfiguration of run-time components and infrastructure; tying the design of software with the traits and properties of the

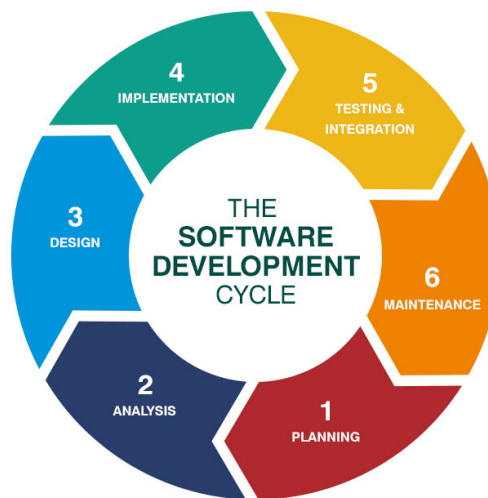


Figure 2.1: Software Development Life Cycle example [Hus16].

infrastructure to be employed [FD14].

In the end, the traditional distinctions between implementation, operation, and design are prone to disappear or be radically redefined. Such change is to be exacerbated with the rise of automatic program repair solutions, due to its insertion in the software development process and the expected reduction of development and debugging time.

2.3 Live Programming

In this world of instant feedback and knowledge at our fingertips, software development has also delved into the immediacy of instructions and/or suggestions. Social media, however, a great example of such feedback, has effects that are of questionable consequences, but, for most intellectual activities, this instantaneous feedback has been proven to be superior to delayed feedback or its absence, even increasing the gain of expertise [KKH13, KA72, EKS93].

At the same time, IDEs used in modern software development have something that many can't live without anymore, code-completion, accelerating code-writing and assisting developers in avoiding typos and easy to overlook mistakes. However, most of these systems are ad-hoc, incomplete, and propose mainly static suggestions of the programming language, and, therefore, its suggestions are mainly irrelevant to the specific working circumstance.

In this regard, at the beginning of the 90s, Steven L. Tanimoto proposed a scale to classify the programmer's 'live' feedback with four levels, named the *liveness* scale [Tan90]. *Liveness* is the characteristic associated with the live execution of a program while modifying it and providing feedback to the developer about their development [Tan13]. Culminating in level 4, a program is "*fully live*", permitting edits while running, and updating to the most recent version immediately without interruptions, a very common level to be achieved by current programming environments. Nonetheless, as predicted in Steven L. Tanimoto's paper, additional levels of *liveness* were defined, by none other than himself, adding two extra levels, totaling the six levels, which can be seen in Figure 2.2,

that defines the degree of 'live' feedback given to the programmer. The new level five, named *tactically predictive*, is not solely constantly running, but also predicts future programmers' actions with multiple alternatives. On the final level, named *strategically predictive*, predictions are strategically analysed to take into consideration every possible desired behavior of a larger software unit.

However, live programming is not all well and good, as it has its handfull of criticisms such as its higher computational burden on the system, or the lack of need for executions for every little change. Many of such criticisms were, though, already addressed by S. Tanimoto, with the aforementioned being from the time-frame between executions ability to be manipulated according to the needs of each project, and the resources required, which are no longer as unwieldy as they once were, though, cases where such is not true, can also be found [Tan13]. Looking at the bigger picture, more recent studies consider that Live Software Development achieves three key characteristics, as per Aguiar *et al*, namely, its **abstraction**, **agnosticism**, and **holism** [ARC⁺19].

The purpose of **live programming** is, then, to break away from the traditional development cycle moving the feedback loop towards a continuum, effectively providing coding conveniences for the developer. The multi-phase development process becomes one, and, from the constantly running program, edits while executing, and debugging within the initial development itself, we arrive at a very different development setting. Such a unique phase minimizes the time between a code change and its respective effect, improves programmers' efficiency and performance, and supports learning with the instantaneous feedback. Ultimately, we can expect to obtain **immediacy**, from the quick overview of the outcomes instead of the cognitive load of *mind executions*, **exploration**, per the trial and error process aided by its immediacy, and **stability**, by creating a bigger awareness of the system's outputs [ARC⁺19].

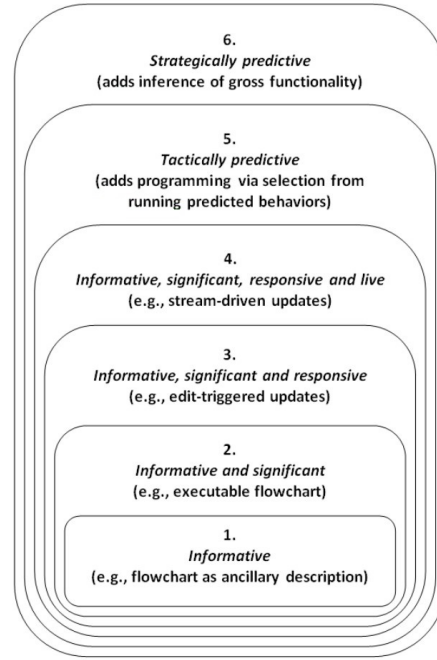


Figure 2.2: Extended version of the liveness hierarchy [Tan13].

2.4 Automatic Programming

Conceived as an issue of the field of genetic programming [OVGB10], automatic programming has become a goal of computer science, and artificial intelligence, since the *birth* of the developer. This concept, in better words, is nothing more than the application of what Samuel has famously said, “tell the machine what to do, not how to do it” [Sam59], to the field of programming. However, even with the current progress of the genetic programming field, including its widespread employment across multiple challenges, automatic programming has stayed mostly unachievable [RW88].

As a whole, the general solution to a computational problem is writing a computer program that solves such a problem. Whereas for researches of genetic programming, the target solution refers to the capability of automatically generate code that solves such problems, in a scalable and independent manner. For this reason, automatic programming is considered to be another AI-complete problem [OS19], entailing that these problems are of so high computational difficulty that only a strong AI would be able to solve. Though, what was once considered a mere construct from an idyllic view of reality has seen small, but significant, advancements [Koz10]. However, recent advancements have shifted focus towards the increasingly higher-level programming languages, which has been helping to drive progress in the field. In other words, the majority of success and focus in the field has not been a true result of automatic programming [OS19], but rather the result of a wider perspective on the problem, that encompasses other fields, such as software engineering, machine learning, compiler optimisations, among others.

In other words, “the dream of automatic programming has eluded computer scientists for at least 50 years” [FNWL09], shifting the focus towards automatic program repair. And even though the automatic program repair field does not create programs from the ground up, they do generate, from existing buggy programs, fixes capable of repairing existing faults.

2.5 Automatic Program Repair

Most of the development costs of software systems have been proven to befall post-deployment through maintenance and evolution [MS06] with some reports claiming that program repair (PR) and evolution have dauntingly risen to the highest place in the cost leaderboard of software development [SPL03], therefore, optimizing these are of the utmost importance. This predicament of the never-ending maintenance is caused by none other than **bugs**, the colloquial term for programming mistakes or faults, a task which is highly time-consuming, difficult and tediously manual. Ordinary debugging activities are comprised of **confirmation**, **triage**, and **localization** and, only then, developers attempt to **fix** and **validate** said fix. **Debugging** is the process of identifying faults in programs or to narrow down to a small number of lines where the fault is located. Although tools exist for triage (e.g., [AHM06]), localization (e.g., [JH05, SND⁺11]), validation (e.g., [YYZ⁺11]) and even confirmation (e.g., [LNZ⁺05]), repair generation has remained a mostly manual procedure. Like in any manmade creation, software faces multiple faults, which, in turn, lead to invalid values, deadlocks, or even crashes of entire systems. As soon as such issues occur in critical applications, it can lead to great money losses or even human lives [ZC09, BB19]. A wide range of work on their automatic identification and elimination, which has had great advancements over the past decade, has been encouraged by such plight.

Finding and fixing as many bugs as possible has been a battle since the dawn of software and a longstanding goal in software engineering, from which the automatic program failures detection field was born. It is important, however, to clarify the difference between two very closely related, but distinct, fields, *i.e.*, *software healing* and *software repair*, both reactive techniques to failures in program execution [GMM19].

- **Software healing** is defined by the detection of software failures at runtime and deployment of modifications to reestablish normal operation of a given system.
- **Software repairing** is defined by the detection of software failures during development, testing or design, and is applied at the source code level, localizing the bug and applying the *fix* to hasten failures from the same fault.

In its essence, Automatic Program Repair (APR) aims to reduce the time and cost spent during both debugging and development, mitigate the cumbersome task of manual bug fixing and to automate the software development process, with bug fixing suggestions or outright direct fixes. In a more holistic view, as Monperrus as put it [Mon18], "automatic repair is the transformation of unacceptable behavior of a program execution into an acceptable one according to a specification". Consequently, there has been an rampant interest increase on this research topic, as it can be seen by the number of tools developed, but even with the growing number of tools developed (*e.g.*, [WNLf09, LB12, LDVFW12, LNFw12, KNSK13, LR15, LLG16, LR16b, MYR16, XMD⁺17, JXZ⁺18, LKK⁺18, MM18, TPW⁺18, XLZ⁺18, DLTL19, LKKB19, MBC⁺19, SSP19, VAH18, HAM⁺20]), practical deployment remains an elusive goal. One reason for this lack of industry-wide adoption is likely to be the rather limited types of bugs that current state-of-the-art APR approaches can properly fix [LFW13, ZS15]. Furthermore, some approaches (*e.g.* [LDVFW12, KNSK13]) seem to only be able to fix elementary bugs, due to various limitations, and most of them produce machine-generated repairs which are unreadable, and of questionable maintainability, *i.e.*, unnatural.

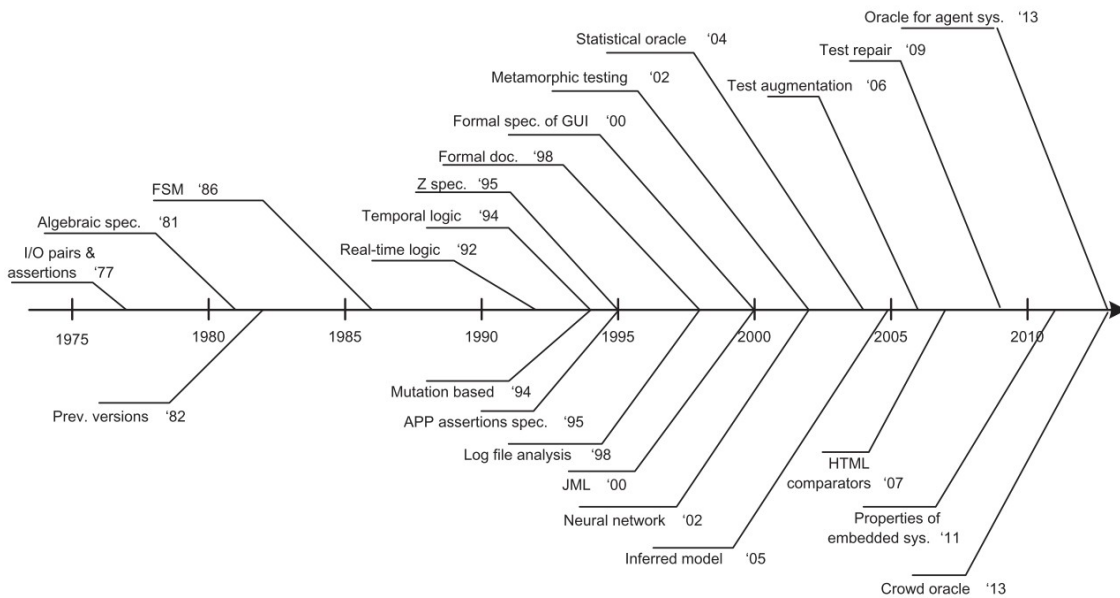


Figure 2.3: Timeline of the evolution of the research on test oracles [PZ14].

However, from some substantial recent work, the utopic world of automatic program repair has never been closer to materialization. For these APR solutions, countless uses can be found, from automated continuous integration bug fixing, outright hint generation for the programmer, or

even at fixing specific categories of security vulnerabilities, specifically, buffer and integer overflow [GPR19], *e.g.*, at repairing the well-known Heart-bleed vulnerability [MYR16]. In short, existing repair techniques usually fall within the following categories, which will be further detailed ahead:

- **Generate and validate repair methodology**, also named, failure-driven [Gin19], test-based [XLZ⁺18] or heuristic repair [GPR19] (*e.g.*, [LNFW12, LKK⁺18, JXZ⁺18, LKKB19]), generate, within a search space, a large pool of possible **repair candidates** and then look for the correct repair using optimization functions within said search space.
- **Semantics-based repair methodology**, also named, constraint-based repair [GPR19] (*e.g.*, [MYR16, XMD⁺17, LB12]), take advantage of program synthesis and constraint solving to generate repairs using semantic information, *i.e.*, by satisfying said constraints derived from the provided test-suites and symbolic execution.

Both techniques may be improved by the use of machine learning, commonly referred to as learning-aided repair [GPR19]. In the end, studies and software repair techniques are various and very recent, innovation and optimizations are being introduced every once in a while and, accordingly, their results are very scattered and complicated to consolidate into a clear and better understanding of the problem as a whole.

Throughout our work we will be mentioning a lot of APR specific vocabulary, which we will further explain below:

- **Bug** - Even though literature is very sparse and confusing, using terms such as *errors*, *failures* or *faults* for the same concept [Mon18], what we will mostly refer to will be *bugs*, *i.e.*, the root cause of errors, and, therefore, faults and programming mistakes will also be used to refer to these. A bug is, therefore, a deviation from the expected behaviour.
- **Specification** - Set of expected behaviours. These may show up as tests, formal specifications, natural language, formulas, *etc.*, and evaluate acceptability, expectation, correctness, and, sometimes, naturalness. A specification is then comprised of several oracles.
 - **Test-Suite** - Input-output-based specification, built from a collection of test cases to specify sets of behaviors from detailed instructions or goals for each.
 - **Pre and Post-Conditions** - Pre-conditions validate parameters at the start of functions before any other code is executed, while post-conditions validate the return value and output parameters. This is a common specification within programs in Java.
 - **Abstract Behavioral Models** - Model with a clear and simple concurrency that allows "synchronous, as well as actor-style asynchronous communication" [Häh13]. These models abstract away from I/O implementations and specific datatypes, while also featuring code generators. Abstract behavioral models feature formal semantics and were designed to reach high formal analysability.
- **Oracle** - Small *gate* that decides if the execution fulfills the pre-defined assertion. Occasionally, oracles are associated to their state in the evaluation of said program, *i.e.*, *bug oracles*,

if referring to the failing test cases, or *regression oracle*, if referring to passing test cases [SWH11].

- **Repair Space** - Collection of generated candidate patches, *i.e.*, candidate modifications to the program.
- **Repair Location** - Statement or group of statements to be modified to repair the program, which may or may not be the buggy location [SSP19].

2.5.1 Generate and Validate Program Repair

A typical generate and validate repair is comprised of three steps, where the first step is **fault localization**, whereby providing a test-suite with both passing and failing test cases, can select a subset of a program's sections with probable bug locations, commonly using spectrum-based fault localization (SBFL) such as Tarantula [JH05], Zoltar [JAV09], and Ochiai [AZvG08]. With this narrowed down search space, we can more efficiently pass to the second step, **candidate patch generation**, which with predefined patch ingredients, a group of possible alterations that form a candidate patch, produces potential candidate patches, *i.e.*, mutations. Since the perfect repair schema is yet to be found, these can be generated through random choice, genetic algorithms, heuristically, or with deep learning models. Finally, **patch selection and validation**, since plausible repairs can be incorrect, is responsible for evaluating each of the candidates, outputting one or more possible patches. Validation is then performed by, usually, software-testing, the most used validation technique, that puts candidate patches against the subset of tests in which the fault resides in, and then, in theory, the complete suite, to ensure that repairs do not unintentionally break other of the program's functionalities. Despite plausible repairs often passing tests, they may sometimes be unable to find a correct repair to most bugs, as it can be seen by most generate and validate APR solutions. Moreover, since testing is computationally expensive, especially, with a large number of tests and patches to be validated, mutation analysis can become troublesome, limiting performant feedback. The candidate patch generation can be, commonly, performed in **three different change operators**. It should, however, be noted that as Gazzola *et al.* [GMM19] have established, generate and validate activities may be carried out following two major strategies, *search-based* and *brute-force*, and, although the whole repair has been identified as *search-based*, herein will use such term uniquely to classify only the subclass that uses search algorithms. In a nutshell, *search-based* change operators via a heuristic search algorithm or randomly, while *brute-force* strategies systematically generate every feasible alteration that can be produced within the search-space while following a limited set of change operators and manipulations.

2.5.1.1 Atomic Change Operators

The atomic change operator applies changes in solely one location of a program's Abstract Syntax Tree (AST), *e.g.*, inserting, modifying or deleting a certain statement or operator of an expression. Since the atomic change operator requires an analysis of a specific location within the program,

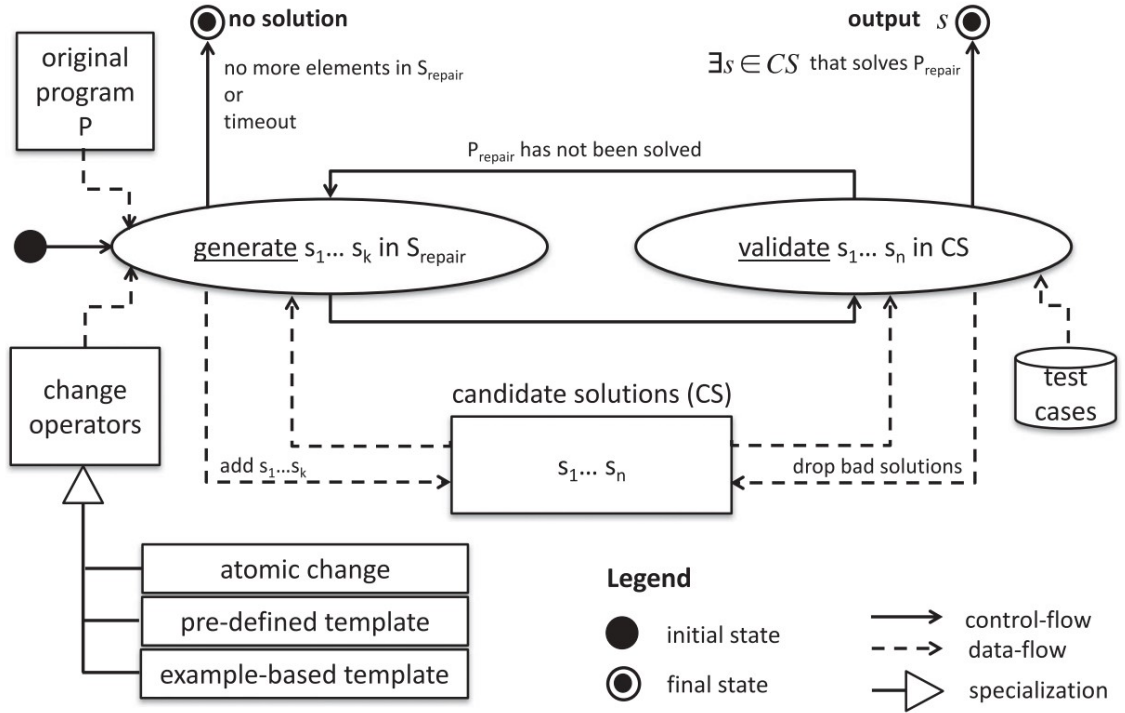


Figure 2.4: Generate and Validate repair process [GMM19].

instead of full program analysis, it is considered to be the simplest of the change operators. Due to its simplicity, these operators are applied more actively and efficiently, generating numerous variants of the to be repaired program, increasing the odds of finding a fix. Throughout existing APR tools that use atomic change operators, many differ in their algorithm and heuristic implementations with varying levels of success.

2.5.1.2 Pre-Defined Templates

Change operators that derive from a set of pre-defined templates can alter one or more statements, by, for example, defining complex change patterns, instead of obtaining such by combining a random amount of atomic operators. Such changes include templates that "expand synchronization blocks, perform non-trivial manipulations on program conditions, and add code implementing predefined access control policies" [GMM19]. Pre-defined templates are mostly explored with *brute-force*, due to the high cost of applying templates, compared to atomic changes, with more *search-based* algorithms.

2.5.1.3 Example-Based Templates

Example-based templates, also named history-driven program repair [Le16], try to address the inaptitude of APR techniques purely based on common test-suites by generalizing many of the features already implemented [LR16a]. Such a feat is achieved by using bug fixing history to compose and employ repair templates as main guidelines for assessing repair candidates and

their quality, unlike the previous techniques, which girded to test suites as the only inputs. This employment may be achieved by a heuristic evolution of said programs (*search-based*), or by constantly performing changes to them (*brute-force*). The concept behind this technique is based on the fact that current bug fixing is often similar, in nature, to past fixes, serving as a guide to the new fault fixing.

2.5.2 Semantics-Based Program Repair

On the other hand, semantics-based repair focuses on fixing smaller and less generic bugs, such as if-conditions and assignments, by devising a constraint that a patched section should satisfy, which is treated as a function to be synthesized. These repairs are, therefore, easier to find, since the technique does not try to produce a complete formalization of the repair problem, *i.e.* rendering the search space more amenable, but rather, solely fix a very specific (set of) characteristic(s).

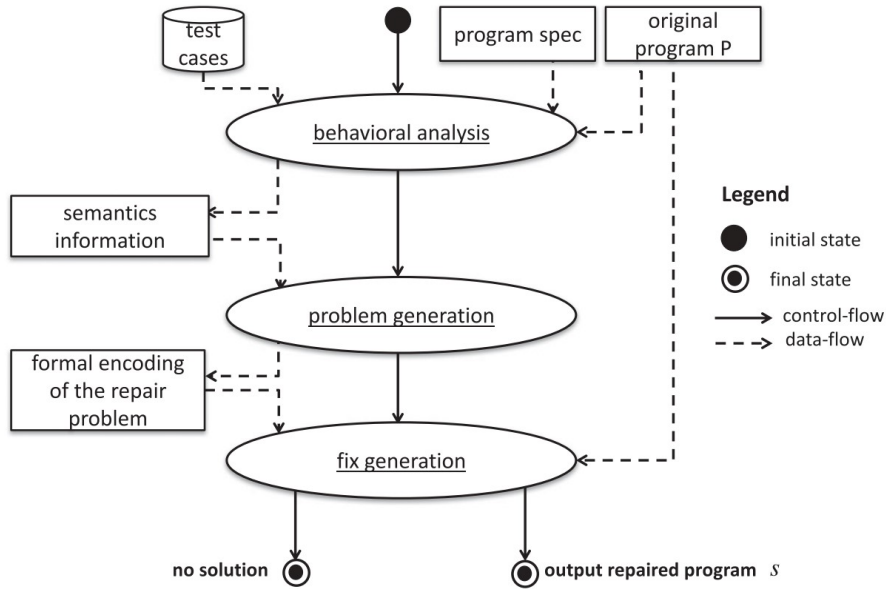


Figure 2.5: Semantics-driven repair process [GMM19].

Like in Generate and Validate PR, the semantics-based PR encompasses three main sequential activities [GMM19] and is initiated by the *behavioral analysis*, which draws out semantic information of proper and improper behaviors of the program. Such activity might explore a subset of all specifications, test cases, source code, etc., to further devise what is deemed correct or faulty and to reach conclusions on what to eliminate or modify. Secondly, the *problem generation* phase begins which is responsible for exploiting the collected information to produce the actual constraints for the problem. With such constraints defined, the next activity will be the *fix generation* which is in charge of solving the actual constraints and generating a proposed code change, if possible.

Thus, in semantics-based repair, the actual formulation of the constraints is key, instead of the solving mechanism, and its efficiency is higher when compared to search-based repair, however, its effectiveness is lowered by its reduced capabilities and hard to solve constraints [Le16].

2.6 Property-Based Testing

Complexity in large software systems has always been somewhat of guaranteed, and with the ubiquity of software development, larger systems are turning into the norm. Such is a sage assumption, as Software is one of the most crucial and complex components of any sophisticated product or service, and following Lehman's laws of software evolution, when functionality grows, there is a matching decrease in quality and increase in complexity [Leh80, LR01]. Therefore, to match the increasing complexity, methodologies to ensure and preserve software quality, robustness and maintainability have been a prevalent topic of research of the latest years. Testing has been, considerably, the most commonly used approach to ensure these requirements, however, it can be highly time-consuming, and, consequently, costly. There have been, however, software development guidelines that have been moving towards functional programming and type-checking to augment trustworthiness in software, though, in practice it is not enough.

As test-based specifications towards automatic program repair typically rely on common TDD, *i.e.* it uses example-based testing, requiring long test generation sessions, where each input scenario has to be considered by the developer itself. Such endeavour is, in on itself, already troublesome, since, as one can imagine, no developer can think of all test cases possible of obtaining different results. There will *always* be that one case that fails the tests, but the developer didn't think of it. On the other hand, **Property-Based Testing** (PBT) [FL94, FB97] is generative testing, *i.e.*, unit tests containing specific example inputs with expected outputs need not be provided, instead, properties are defined and a generative-testing engine creates randomized inputs to verify such properties. However, in early development, it is sometimes hard to envision properties and example-based testing does provide for better feature compliance, therefore, there is still a place in the software development cycle for the run of the mill unit tests. These later help serve as anchor points for the development of property-based tests, which will cover a much wide input scope. Taking this into account, we believe that the use of property-based testing as specifications towards automatic program repair is capable of augmenting the number of bugs found, and, in turn, augment the reliability of patches.

Furthermore, property-based testing frameworks require only a very small number of lines of code, and test a different set of inputs each time, covering a larger domain space with a sizeable difference from unit tests with the same amount of code. Even though the learning curve may seem steep at first, thinking of properties instead of example-based test demand more cautious thinking, and, thus, results in a better result [CH00], *i.e.* the initial investment may seem high, but the reward is greater. Usually, these frameworks run hundreds of different inputs, and try to fail the testing using a myriad of values from empty lists, negative values, and other possible edges cases. Furthermore, it also passes high numbers, long lists, and strings with special characters. For cases when users have custom data structures, most PBT frameworks also provide custom generators, to allow for the random generation within a given data structure. Moreover, shrinking, *i.e.*, shrinkage of large inputs that failed for better understanding and reproducibility, and race conditions detection are common features in PBT suites. In sort, properties provide more a concise and simpler approach

to the maintenance of test suites, and helps build a better test coverage [LTA11].

Chapter 3

Problem Statement

3.1 Overview	19
3.2 Issues	20
3.3 Hypothesis and Research Questions	22
3.4 Validation Methodology	23
3.5 Proposal	23

We start this chapter by describing the problem that exists within the current automatic program repair solutions (*cf.* Section 3.2, p. 20). We then proceed to give a brief overview of the listed issues (*cf.* Section 3.1), leading up to our research questions (*cf.* Section 3.3, p. 22). Finally, based on the state of the art (*cf.* Chapter 2) and the listed issues we present an initial proposal (*cf.* Section 3.5, p. 23).

3.1 Overview

With the rising complexity of software systems, automated program repair solutions have seen a growing interest by the industry to account for higher demands in quality, performance, robustness, and maintainability. With that in mind, the scientific community has developed a myriad of studies and tools to address this demand, however, their use is still crippled by some very pressing issues. Individually, these issues are solved by many tools and solutions as mentioned in Chapter 2 (p. 7), but as a whole, the research field is still far from creating a true panacea capable of solving all of them. Furthermore, these tools have predominantly been restricted to academically oriented applications (as only around half of the techniques have a tool available on the Web [GMM19]), therefore IDE integration with patches as live feedback has been a rare sight only fulfilled by AutoFix [PFNM15] and ccheck [LB12], with the most recent implementation by Campos [Cam19]. These integrations allow for a speedup in development and an increase in productivity [BMM09], wherefore the latter has implemented an efficient, practical, error preemptive and tactically predictive [Tan13] automatic program repair tool. Though, several issues were put aside by Campos' thesis, in particular, its

completeness, overfitting and adaptability to other IDEs. Furthermore, this implementation lacked the support for multiple errors, always presenting the user with a single suggested fix at a time, *i.e.* the most recent suggested fix found, instead of all of them, limiting the amount of information the user could get. In this work, to achieve specification completeness, the tests' creator has to think of all the possible cases, though if such test cases can be thought of, they are probably already well regarded during their development, ending up with success during the testing phase. Therefore, there is a need to find the test cases that the test creator cannot reach, as it has been proven that automated test case generation can be helpful in the domain of program repair [MYR16, SBLB15, LLL16]. Additionally, APR solutions have not yet solved the matter of overfitting, as most patches, when complete, are overly tailored to specifications, failing to identify the problem instead of to the solution.

3.2 Issues

Even with all the automatic program repair (APR) solutions developed in the past decade, we can still find several issues currently affecting most APR solutions [LR16a, QLAR15]. For now, APR has been behaving as a hydra, where whenever we cut one obstacle off, two more would grow back, which, in a sense blocks us from ever achieving perfect APR. For this reason, it is widely considered that automatic programming is another AI-complete problem [OS19], entailing that these problems are of so high computational difficulty that only a strong AI would be able to solve. To put it in other words, we first must crawl to learn how to walk.

Future automatic program repair solutions should then focus on the following sections problems:

Automatic Program Repair Issues

1. **Error preemption** - Using APR solutions to fix bugs introduced in the past has been the sole focus of APR solutions for a long time, however, providing developers with the tools to build programs without errors in the first place is still a rather untouched research field.
2. **Generality** - Since the amount of programs and bugs that tools are able to fix is rather small [LFW13], a broader application of these solutions has been seen to not counterbalance the risk.
3. **Credibility** - Since programmers have yet to develop trust in these tools, mostly due to a lack of understanding of their inner workings, credibility is one of the problems currently affecting most APR tools' real-world practicality.
4. **Scalability** - Scalability is one of the most impactful issues in this purpose since only a very limited number of tools can find repairs within a short timeframe [GPR19, HO18, MYR16, GGM19].
5. **Maintainability** - As many of the existing approaches follow a stochastic approach, and even those that do not, fixes produced often are not as robust as a manmade fix would be, often

generating machine-generated repairs which are unreadable, unsound, and of questionable **maintainability** [MYR15, CSC⁺19].

6. **Efficiency** - Many of the current techniques either try to find either minimal patches or try to broaden the possible patches, minimizing the odds of finding and fixing as many patches [QML⁺14, LCL⁺17, SL18, TPW⁺18, YB18].
7. **Completeness** - The strength of the oracles and their completeness are heavily dependent on test suites, however, there is still no well-defined method for creating test suites capable of enhancing APR solutions' capabilities [ZZH⁺19].
8. **Overfitting** - Patches produced by many of the APR tools available face one of the biggest challenges in this research field, patch overfitting, that is, patches produced are tailored to the specific test data provided, sometimes even deleting functionality, greatly reducing trust in integrating APR solutions in automated processes. Simply put, produced patches may be incorrect since they are not automatically verified for correctness, instead they still heavily rely on developers' expertise to assess correctness, which, as humans, may fail. Therefore, it is of the utmost importance that suggested patches to not crumble to overfitting [SBLB15, MM18, KLB⁺19].

Investigation Issues

- **Formalizing patch quality** - Patch quality is one of the biggest hurdles in the practical adoption of APR solutions [LFW13], and its reliability. Often characterised as patch **soundness** or **naturalness**, patch quality is a metric that has yet to be defined and specified, therefore, it forbids most APR techniques of becoming fully automated, demanding surveillance from human developers. Thus, the ability to quantitatively measure, predict, and ensure functional patch quality to then present humans with only the repairs that provide the highest quality standards is an important step towards improving the current state of APR techniques. However, such is still farfetched as it would likely require studies to comprehend and measure the factors that affect programmer understanding of a patch.

Technical Applicability Issues

1. **IDE integration** - The integration of APR solutions within IDEs is certain to help developers avoid programming faults altogether, by presenting the developer with live feedback. This accelerates the time to fix, since the need for the developer to analyse previously pushed code that one might not recollect the exact functioning disappears, and, instead, suggested fixes are presented throughout the development. Such has been proven by simple code completion tools that display a level 5 in the liveness hierarchy, effectively providing "tactically predictive" feedback [Tan13] which has been proven to speed up development and increase productivity [BMM09].
2. **Fix Cross-Validation** - Due to many of the issues listed above, fixes produced by automatic program repair techniques generally need to be validated by developers before being

integrated by into the codebase. While some fixes might be checked trivially, many are non-obvious and demand a particular degree of adjustment to be accomplished. For this reason, although fixing faults automatically may seem extremely cheap, it is a naive assumption ignoring the true cost of most automatic repair processes, as it does not take into account the developers' effort in inspecting outputs produced by software repair techniques.

3.3 Hypothesis and Research Questions

The product of this dissertation is founded in a hypothesis that sets the objectives for both its development and implementation, which is:

“Using Property Tests as Specifications in an Automated Program Repair tool helps to eliminate overfitting.”

Such a statement can be subdivided into different specific elements, serving as the building block of our hypothesis, *i.e.* the research questions:

- **Research Question 1** - *“Are developers capable of understanding and formulating property-based tests?”*
- **Research Question 2** - *“Do developers believe in the completeness of example-based tests?”*
- **Research Question 3** - *“Can property-based testing truly solve overfitting?”*
- **Research Question 4** - *“Do developers believe a live Automatic Program Repair technique is easy to use?”*
- **Research Question 5** - *“Can developers see a positive impact on their workflow through the use of a live Automatic Program Repair technique?”*
- **Research Question 6** - *“Do developers trust the capabilities of a live Automatic Program Repair technique?”*

Furthermore, as validation of both Campos' work [Cam19] and our own, we have decided to take a deeper look at the research questions of the author:

- **Research Question 7** - *“Are users faster in reaching the solution when using a live Automatic Program Repair tool?”*
- **Research Question 8** - *“Are solutions generated by an Automatic Program Repair tool different from the ones developed by human programmers?”*
- **Research Question 9** - *“Are users aware of the rationale of solutions generated by the Automatic Program Repair tool before accepting them?”*

3.4 Validation Methodology

In this dissertation, to validate the main hypothesis, declared in the section above, a case study will be defined, to be experienced by different participants in different scenarios. Science, as a field based on experimental validation and repetition, allows us to corroborate hypotheses, assumptions, and conjectures with studies such as this one since it has been proven again and again that such is also the case of Software Engineering [FPG94]. Taking this into account, a controlled experiment was envisioned to validate the hypothesis and research questions presented in Section 3.3. This study will consist of evaluating the performance of developers when using the implemented Visual Studio Code extension with properties as specifications and comparing it to the performance when not using properties as specifications, by measuring if the overfitting caused by incomplete example-based techniques can be eliminated with property-based testing.

With this objective in mind, we will define objectives of the validation (*cf.* Section 5.1, p. 45), guidelines over which the study will conduct itself over (*cf.* Section 5.2, p. 46), further planning of the empirical evaluation (*cf.* Section 5.3, p. 47), tasks to be performed (*cf.* Section 5.4, p. 50) and, with such, the respective results obtained and their extensive analysis (*cf.* Section 5.5, p. 55). From the results, we intend to extract important and highly pertinent data which may help us reach conclusions about the proposed solution and our hypothesis (*cf.* Section 3.3, p. 22).

3.5 Proposal

As it has been noted, most APR solutions have demonstrated to have a very important limiting factor, test suites, which may drop functionalities due to weak test suites, or, on the other end, make fix generation too complex due to the strong test suites. Our focus for this work will mainly revolve around **eliminating overfitting**, while introducing **completeness, soundness, and error preemption** with a true **real-world practicality**, by integrating it as an extension to the most popular IDE whilst using a modular architecture capable of easy adaption by other IDEs. Notwithstanding, in our work we intend to implement and integrate the world of property-based testing within the automated program repair research space, effectively *killing two birds with one stone*, by taking advantage of APR solutions to suggest semantic suggestions to the developer while using properties to define specifications. This solution, like its predecessor, implemented by Diogo Campos [Cam19], would introduce the repair process during development instead of after, combining the advantages of both the developer’s reasoning and the machine-generated repairs, as repairs would be merely suggestions to improve the developer’s thought process.

As A. Marginean *et al.* have put it, developers are overall trusted in their abilities, therefore developers are a highly useful final oracle [MBC⁺19], which, for now, seems to be a good ultimate defence against faults after passing the required test suites. Nevertheless, the current state of the automatic program repair research space requires such, as even the most efficient solutions do not provide all solutions with naturalness, maintainability and soundness of a human developer. Though, the necessary effort to cross-check and comprehend the suggested fixes might be so significant,

that it may potentially mitigate the usefulness and help of automatic techniques. Thus, by bridging the automatic repair world with the liveness areas of research, we create a light-weight solution capable of helping developers fixing their bugs, instead of doing it for them facilitating and assisting the developer during the development of the code, ensuring better code quality, maintainability, soundness and faster time to fix. Finally, we will try to empirically validate if we can overcome the overfitting issues presented in current APR solutions by applying property-based testing to the world of APR, and with the data gathered, answer our research questions.

Chapter 4

Proposed Solution

4.1	Contextualization	25
4.2	Objectives	25
4.3	Implementation	27
4.4	Summary	43

This chapter will detail and elaborate on the proposed solution to solve the problem stated in Chapter 3, firstly contextualizing the previous developments on Live automatic program repair (*cf.* Section 4.1). Subsequently, we will enumerate and analyse the objectives of this research project (*cf.* Section 4.2) and also detail the implementation of the proposed solution (*cf.* Section 4.3, p. 27).

4.1 Contextualization

This dissertation is the evolution of a tool previously implemented by Diogo Campos [Cam19] and is part of the research in live software development that is being carried out by the Software Engineering group of the Faculty of Engineering of the University of Porto. Even though liveness is an important aspect of this dissertation it also incorporates several other subjects apart from the liveness topic, as noted in Chapter 2 (p. 7). Therefore, this Chapter will briefly cover parts of the previous implementation, but mainly focusing on the changes and improvements made over the previous implementation.

4.2 Objectives

Regarding the development and implementation of the solution proposed with this dissertation, our objectives are well defined and fivefold, which we will proceed to enumerate:

1. **Language Server Protocol (LSP) Implementation** - As it has been discussed, the integration of APR solutions within IDEs is certain to help developers avoid programming faults altogether, by presenting the developer with live feedback, helping with cross-validation that

all current APR solutions require. The previous implementation already had Visual Studio Code implementation due to its popularity, however, it was limited to this IDE. The language server protocol defines a protocol used across IDEs and a language server, which provides features like auto-complete, go to definition, find all references etc. This protocol is highly useful to our case as it allows the servers to be written only once, allowing integration with other IDEs by simply creating a simple client. Even though LSP implementation is a single objective, it encompasses a lot of implementation limitations, architecture changes and a lot of groundwork that requires rethinking and a lot of research.

2. **Greater Mutation Spectrum** - Mutations in previous versions of the tool were very limited, therefore, we wanted to expand the number of supported mutations, in order to augment the odds of finding a solution to a bug found.
3. **Improve Function Types Support** - Due to the language's massive flexibility, in both *JavaScript* and *TypeScript*, functions can be defined in a number of ways. Previous versions of this tool only supported the "basic" function declaration, however, arrow functions, method declarations, function expressions, and property definitions were not supported, even though they are heavily used within these languages.
4. **PBT Framework Integration** - An important step in the implementation of this revamp of the initial implementation was to ensure property-based testing integration within the tool. When test cases can be thought of during their development, they are likely already defined within the test suite, and, therefore, well regarded within the codebase. However, there is a need to find test cases that the test creator cannot reach, which makes this into a crucial component which will allow us to further validate whether or not PBT will allow us to find the test cases that the test creator cannot reach and, in turn, find more bugs and potentially more bug fixes.
5. **TypeScript Support** - According Stack Overflow's Developer Survey¹ JavaScript still holds the crown of the most popular programming language, being used by 67.5% of respondents, however, TypeScript, an open-source programming language developed and maintained by Microsoft, has seen its utilization rising year over year as also observed by the survey. As a strict syntactical superset of JavaScript that adds optional static typing to the language, it becomes a highly-productive development tool for large applications that transpiles to JavaScript, strengthening its position in the industry. Previous implementations of this extension already took advantage of TypeScript's Compiler API², however, they did not support TypeScript execution.

¹Stack Overflow Developer Survey 2020 (Retrieved by: 28 June 2020)

²TypeScript's Compiler API (Retrieved by: 23 April 2020)

4.3 Implementation

As referred in Section 4.1, we started from a solution that was already implemented, however, this solution required a full refactoring due to several issues as listed in Chapter 3. Considering that the lack of semantic suggestions in code completion tools is a restricting factor in the improvement of developer productivity as mentioned in Section 3.2, we decided to further continue on the path of developing an integrated development environment (IDE) extension.

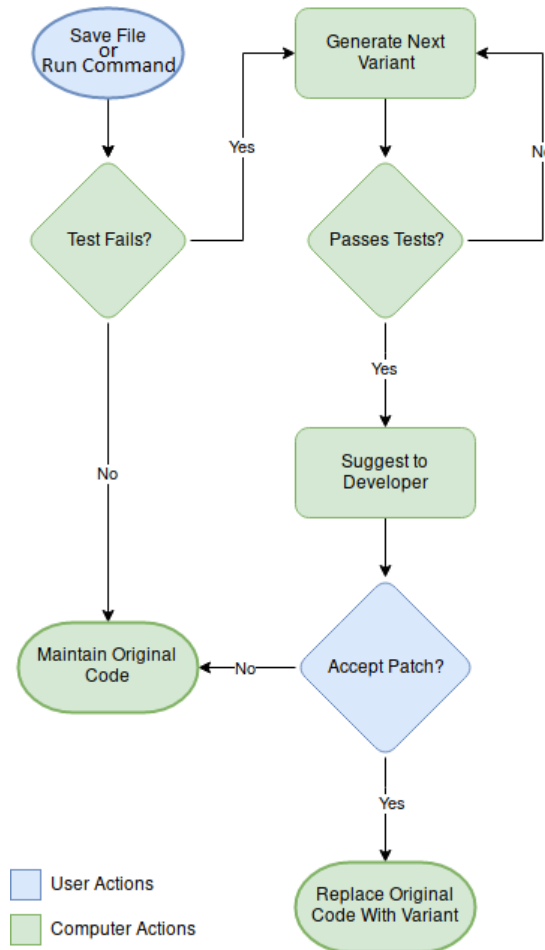


Figure 4.1: Flowchart of the automated program repair process based on the implementation by Diogo Campos [Cam19].

Still appertaining to the assumption that if a program passes all provided test cases then it is correctly patched, we continue on this path of leveraging unit tests as specifications towards generating automatic program repair suggestions. As discussed, the previous implementation had small issues that limited usability, suggestion power and true real-world deployment, but these paled in comparison to the lack of the adoption of the language server protocol, which limited IDE support to solely VS Code. As such, we will further develop on the previous reasonings of leveraging unit tests to provide semantic suggestions as an extension to existing IDEs.

Furthermore, since we were to build on the mutation-based solution, similar to the one implemented by Debroy and Wong [DW10], we mostly maintained the solution's flow of actions, as demonstrated in Figure 4.1 (p. 27). In this figure, we can observe that the whole process starts either initiated by the developer saving the file or by running a command, which runs the existing test suite. If all tests of the said test suite succeed, the process ends, however, if they fail, they start the automatic program repair. As mentioned, our solution is mutation-based, therefore, variations of the files start being generated with said mutations. These variations are then tested and if they pass the test suite, they are considered as possible fixes and to meet the requirements and, subsequently, suggested to the developer as such. The developer considers all the patches, and, if said developer considers one of them acceptable, it updates the source code accordingly. However, suggestions are mere suggestions and the developer might just use them as guidelines to a more tailored approach to the fix.

With this reasoning, we decided to name our tool **pAPRika**, since its function is not to replace the developer, but to help the developer with small hints and suggestions, which we call *spices*. As one can deduce, the capitalized *APR* is an allusion to the automatic program repair research field.

4.3.1 Automated Program Repair

Regarding the automated program repair technique, this one mostly remained unchanged. Thus, to generate valid suggestions pertaining the code under review the following components were taken into consideration.

- **Unit tests as specifications** - Test suites are to be used as specifications, with the purpose of fulfilling the requirements of generating fixes merely based on unit tests, and not in any formal or another kind of informal specification.
- **Generic approach to behavioural repair** - Several classes of bugs must be supported, however, this component was heavily expanded to match with well-know mutation frameworks like Stryker Mutator³.
- **Immediate solution** - Solutions must be found rapidly to ensure the immediate feedback referred in Section 3.1. This valuable real-time feedback to the developer serves to attempt to mitigate manual bug fixing, reduce time and cost spent for debugging.
- **Complete solution** - Solutions presented to the developer, even if, apparently, unnatural, should always be syntactically correct and pass all tests. Partial solutions shall not be presented by the tool and syntactically incorrect fixes rejected.

Given that we opted for continuing the development of a mutation-based solution, such allowed for the use of unit tests as specifications, as the specifications output is the only requirement. Additionally, by employing a heuristic technique for mutant generation, we are able to explicitly

³Stryker (Retrieved by: 28 June 2020)

specify the supported classes of bugs. This amount of control over the number of supported classes allows us to better manage the search-space too better meet the live programming criteria. Our objective for this work will mainly revolve around offering the developer true **real-world practicality**, by integrating it as an IDE extension, therefore while other techniques, such as GenProg [WNL09, LNF02, DTL09], SapFix [MBC⁺19] or Astor [MM18], have a higher probability of finding bugs and higher success rate in repairing them, their performance is highly lacklustre and, thus, impossible to implement on a tool that intends to produce immediate feedback. Ultimately, through the potential solution's requirement of passing all test cases, we can ensure that all suggestions are complete solutions. It is also worth observing that each unit test should correspond solely to a single *unit*, in our case, a single function. Accordingly, every mutation will originate from the *unit* being tested, drastically reducing the amount of statements to mutate, and the required search-space. Throughout this section, we will further discuss implementation details, required changes to the original source code, LSP implementation details, mutation generation, among others.

4.3.2 Language Server Protocol

Before the time of the language server protocol, there was only chaos, and if one wanted to build a plugin for everyone to enjoy, one had to build said plugin for Sublime Text, Vim, VS Code, Atom, etc. Furthermore, language servers, usually implemented in their native programming languages, present a challenge in and of itself, since Visual Studio Code runs on a Node.js runtime, and other IDEs might even have different runtimes, further increasing the chaos.

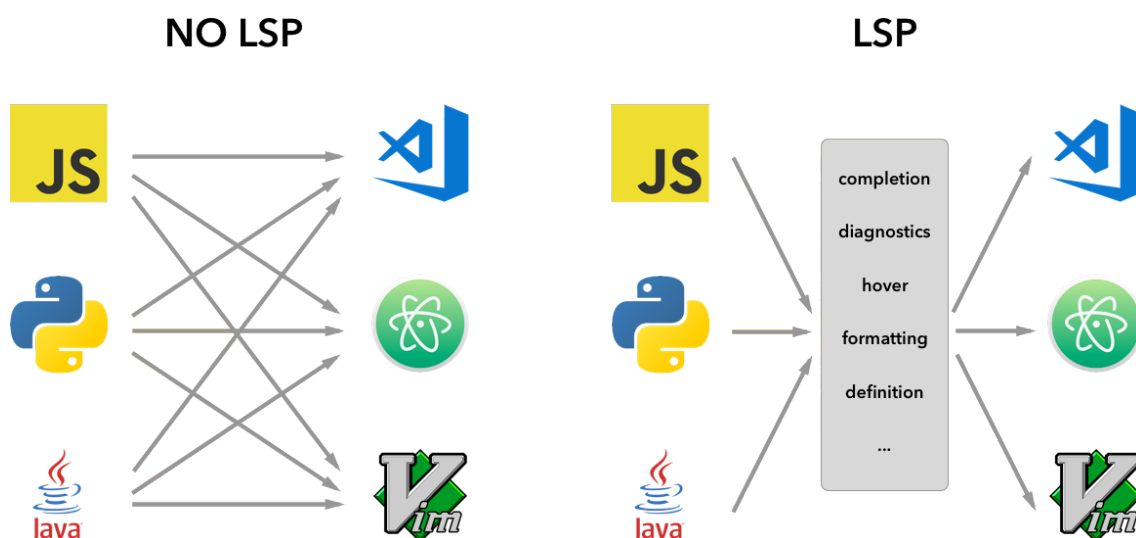


Figure 4.2: Difference between an implementation without LSP versus one with it.

But then, the language server protocol arrived, developed by Microsoft; it aims to establish a standard communication interface for programming language analyzers such as our tool, a protocol between a client, the extension, and a server, where all the processing occurs. With this protocol, we can eliminate the necessity of recreating entire extensions throughout all of the IDEs that are

currently available on the market, and the ones that will emerge later on, as each IDE has their APIs to implement the same feature. Efforts can now be focussed into a single high performant language server, providing code completion, hover tooltips, jump-to-definition, among others, which can now be re-used in multiple development environments with only slight modifications required to work, *i.e.* reducing the implementation cost of these language server extensions for M languages in N code editors from $M * N$ to M .

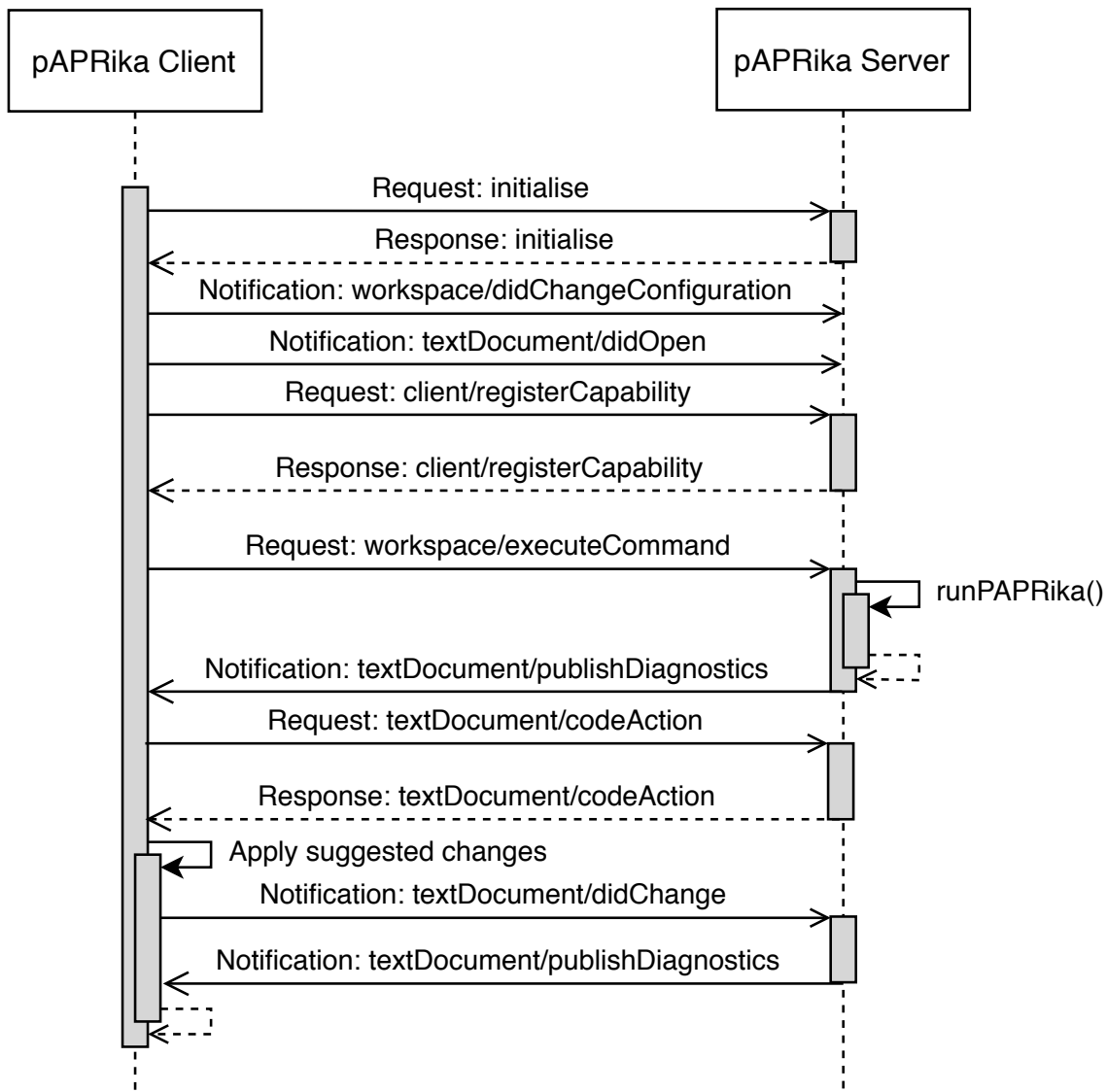


Figure 4.3: Example notification exchange between pAPRika client and server.

The language server protocol essentially works by running a language server as a separate process to the IDE, while using the language protocol JSON-RPC as a means of communication. Although, the pool of available extension features while using the LSP stayed mostly the same when compared to developing an extension solely for one IDE, it enabled a more streamlined implementation of many of them, as well as a more well structure architecture. However, language servers need not to implement support for all supported features by the protocol. Thus, language

servers should announce their capabilities, to clearly define what should and what should not be sent to the language server by the code editor. Figure 4.3 (p. 30) shows an example of the notifications exchange during the execution of our tool, moreover, throughout the Section 4.3.4 (p. 37), we will better clarify some of the supported notifications.

However, this implementation required that some previously supported features were initially discarded due to the lack of access to the VS Code API through the server, such as the support for test case failing identification. Nevertheless, its re-implementation is already planned and thought of, as features can now be implemented faster and with less difficulties thanks to an improved project structure. This feature, however, will have to be developed on the client, as it is a capability that is IDE dependent.

Furthermore, our extension took advantage of the document manager provided by the *vscode-languageserver* module⁴ to incrementally synchronize documents between VS Code and the language server, to avoid the constant large data transfer of the whole content of text documents.

For this dissertation, only the Visual Studio Code client was implemented, however, as mentioned before, the adaptation to another IDE should not require much effort. Furthermore, unlike the previous implementation, the tool is now fully buildable from scratch without any complex processes, facilitating open source development. Such process is facilitated by Visual Studio Code Extension Manager - *vsce* - a command-line tool for packaging, publishing and managing VS Code extensions, that was built into the Continuous Integration/Continuous Deployment (CI/CD) process.

4.3.3 Mutation Generation

As previously mentioned (*cf.* Section 4.3, p. 27), due to the good results obtained by Diogo Campos in his previous implementation, and similar to the one implemented by Debroy and Wong [DW10], we decided to further pursue the mutation-based suggestion formation. Mutation-based approaches require some reflection on which mutations to pursue, as well as the order in which they appear, thus in the previous implementation, when creating the first mutations, the Defects4J dataset [JJE14] was taken into consideration, as it is considered one of the leading databases of existing faults. However, for such task, 3 paramount requirements to ensure the live feedback of the tool had to be defined and taken into account before establishing which mutations should be added to the mutant generation algorithm:

1. **No mutation should increase complexity** - Mutations shall not increase branching, nor should it increase cyclomatic complexity.
2. **No additional statements** - There exist an infinite number of statements that can be added [DW10], therefore mutations shall not affix supplementary statements. Furthermore, to preserve program consistency, statement removal is also disregarded, limiting mutations to adjustments of extant statements.

⁴[vscode-languageserver-node](#) (Retrieved by: 20 May 2020)

3. **One mutation per mutant** - Defining the search-space is the crucial first step to Generate and Validate APR tools as noted in Chapter 2.5.1. Consequently, allowing more than one mutation per mutant, *i.e.* each file generated by the tool may possess more one mutation, is likely to swiftly augment the search-space, which may result in more discovered solutions at the cost of an exponential rise in performance requirements. Thus, restricting the solution to exactly one mutation per mutant generated file will substantially reduce the search-space, and, in turn, the time required for it.

In view of the fact that our solution is mutation-based, possible repair groups need to be analysed and researched upon. The initial analysis of these possible repair action groups relied heavily on the Defects4J dataset [JJE14] and the following Defects4J dissection [SDM⁺18], alongside with its online appendix⁵. In this dissection, *Sobreira et al.* [SDM⁺18] have found and defined nine repair action groups, with each containing three types of operations, *i.e.* mutations: addition, removal and modification. From the nine repair action groups, four were ruled out from consideration for the developed tool, for the following reasons:

- **Method definition** - Every mutation will originate from the *unit* being tested, *i.e.* mutations are solely applied within the function definition, which drastically reduce the search-space, therefore no method definition mutations are considered.
- **Exception** - We have removed this repair action group due to the *no additional statements* constraint, as all patches within this group are tasked with adding or removing statements.
- **Type** - In the previous iteration of this tool, this repair action group was removed since JavaScript is a dynamically typed language and not a statically typed one, however, in the current version, since TypeScript is supported, such was not a valid argument. However, replacing variable types represent a potentially large number of variations, therefore it stayed excluded from this development work. Though, we are open to change this in a future release of our tool.
- **Variable** - The only mutations that only require modifications in this action group are, firstly, type and modifier changes, which only have a presence in TypeScript, however, since our mutations rely solely on the AST inside a single function, such would have no effect in the global scope. And, secondly, replacing variables by other variables or method calls, which by itself presents a potentially infinite number of possible variations and is therefore discarded, as it goes against our *immediate solution* requirement.

We then analysed the remaining five groups, though, only modification operations were considered, where the respective mutations to implement were the following:

- **Assignment** - Modifications on the Right Hand Side (RHS) of assignment statements.

⁵Defects4J Dissection (Retrieved by: 30 June 2020)

- **Conditional** - Modifications on the RHS of conditional expression statements.
- **Loop** - Modifications on RHS of initialization variables and modifications on the RHS of conditional tests.
- **Method call** - Method call moving and modifications on parameter values.
- **Return** - Modifications in return expressions.

These modifications were initially achieved by Switch Mutations, Parentheses Mutations, Off-By-One Mutations, Operator Mutations and Statement Moving Mutations, however, some of these mutations were incomplete, required refactoring, namely the Operator Mutations. New mutations were also implemented, such as Boolean Mutations and Remove Prefix Mutations, based on the list of Stryker Mutator's supported mutations⁶. This list was also the inspiration for some of the required improvements to the already implemented mutations.

The following sections will describe in more detail the several supported mutations, which will be demonstrated with pseudocode adaptations and replacement tables. These mutations are performed in two steps by our algorithm, initially examining the Abstract Syntax Tree (AST), generating mutations when feasible, using the supported mutations for each case. Thereafter, the heuristic performs the statement switching, iterating through each line of code, between the current and following line, resulting in all the possible Statement Moving Mutations. As mentioned in the previous section (*cf.* Section 4.2, p. 25), all mutations were performed using TypeScript's Compiler API⁷, which provides access to multiple tools to interact with both JavaScript and TypeScript files, since the latter is a superset of the former.

Though, mutations are not applied randomly, as they are only applied in specific nodes in the AST. The target abstract syntax tree, *i.e.* of the function being tested, must firstly be found. This function, in the previous implementation, had to be a "*basic*" function declaration, however, in our implementation, has expanded its support to arrow functions, method declarations, function expressions, and property definitions, which required an upgrade to the AST seeking algorithm, and has a more in-depth look can be found in Section 4.3.4.1 (p. 38).

For each mutation, we will identify the node types it applies to, which we will proceed to list:

- **Binary expression** - These nodes represent a specific type of binary tree that represents expressions, which are mostly algebraic. Binary tree nodes, and hence binary expression tree nodes, have zero, one, or two children, representing: Left Hand Side (LHS), Operator and RHS, which in pseudocode algorithms we will be represented by the properties *lhs*, *op* and *rhs*, respectively.
- **Identifier** - Identifiers are symbols, *i.e.* tokens, which name language entities such as variables, namespaces, types, classes, methods, interfaces, constants, macros or parameters.

⁶Stryker Handbook - Supported Mutators (Retrieved by: 29 June 2020)

⁷TypeScript's Compiler API (Retrieved by: 23 April 2020)

Identifiers are used to uniquely identify program elements in code, however, in our mutations, the important factor is to know their location, and not what they define.

- **Variable declaration** - This node represents nothing more than the declaration of variables, containers for storing data values. Such declaration attributes a name and data type to a variable, which can happen in the form of *var*, *let*, and *const*.
- **Element access expression** - This node represent the element access of an array or indexer.
- **Prefix unary expression** - As its name implies, this is an operation with a single operand, in contrast to binary expressions, which use two. For our tool we only implemented prefix mutation, as postfix and functional expressions do not usually represent common bugs in code.
- **Boolean** - This node is nothing more than a boolean data type, which has one of two possible values (usually denoted true and false). Mutations to this node are useful for generating boolean variants as we will discuss below.

Every mutation, when created by the mutation algorithm and before being tested, was added to a list of mutations, the *ReplacementList*. This internal representation of the list of possible changes was comprised of several *Replacements*. A *Replacement* is an internal re-implementation of TypeScript API's *TextChange*, which on previous versions only stored both the old and the newly generated node. However, that information was lost as soon as the testing phase finished, which meant that when the developer hovered the faulty code, and the code editor asked for a Quick Fix (*cf.* Section 4.3.4.3, p. 40) the extension did not know what the replacement was. The extension, then, had to process the suggestion message, and attempt to parse out the new text from said message, which led to several cases of wrong fixes and absurd formatting caused by the tool.

In lieu of this, we implemented a key-value pair for each document, where its value was another key-value pair for each complete suggestion generated and sent to the developer (identifiable via a *replacement code*), with the respective *Replacement*, *i.e.* a Map of Maps with all the complete suggestions. Such has allowed for deprecation of the suggestion message parsing, and, instead, faster and more reliable suggestions.

4.3.3.1 Switch Mutations

The Switch Mutation consists of switching the left-hand side and right-hand side of binary expressions, maintaining the operator, as presented in Algorithm 1.

Switch Mutations are applied to every **binary expression**.

Algorithm 1: Generate Switch Mutants

```

1: procedure GenerateSwitchVariants(node; replacementList)
2: newNode  $\leftarrow$  new BinaryExp(node.rhs; node.op; node.lhs)
3: replacement  $\leftarrow$  new Replacement(node; newNode)
4: replacementList.push(replacement)

```

4.3.3.2 Parentheses Mutations

The Parentheses Mutation consists of two variants where both the left-hand side and right-hand side of binary expressions get parentheses added, maintaining the operator, as presented in Algorithm 2.

Parentheses Mutations are applied to every **binary expression**.

Algorithm 2: Generate Parentheses Mutants

```

1: procedure GenerateParenthesesVariants(node, replacementList)
2: newLhs  $\leftarrow$  AddParentheses(node.lhs)
3: newRhs  $\leftarrow$  AddParentheses(node.rhs)
4:
5: lhsVariantNode  $\leftarrow$  new BinaryExp(newLhs, node.op, node.rhs)
6: rhsVariantNode  $\leftarrow$  new BinaryExp(node.lhs, node.op, newRhs)
7:
8: lhsReplacement  $\leftarrow$  new Replacement(node, lhsVariantNode)
9: rhsReplacement  $\leftarrow$  new Replacement(node, rhsVariantNode)
10:
11: replacementList.push(lhsReplacement)
12: replacementList.push(rhsReplacement)

```

4.3.3.3 Off-By-One Mutations

The Off-By-One Mutation consists of generating two variants where the right-hand side of a binary expression is modified, one by adding 1 and another by subtracting 1, as demonstrated in Algorithm 3 (p. 36). This mutation maintains both the operator and the left-hand-side. Another variation of this mutation happens in identifiers, where we also generate two variations for the whole node.

Off-By-One Mutations are applied to every **binary expression**, **identifier**, **variable declaration** and **element access**.

Algorithm 3: Generate Off-By-One Mutants

```

1: procedure GenerateOffByOneVariants(node, replacementList)
2:   minusOneRhs  $\leftarrow$  node.rhs - 1
3:   minusOneNode  $\leftarrow$  new BinaryExp(node.lhs, node.op, minusOneRhs)
4:   minusOneReplacement  $\leftarrow$  new Replacement(node, minusOneNode)
5:
6:   plusOneRhs  $\leftarrow$  node.rhs + 1
7:   plusOneNode  $\leftarrow$  new BinaryExp(node.lhs, node.op, plusOneRhs)
8:   plusOneReplacement  $\leftarrow$  new Replacement(node, plusOneNode)
9:
10:  replacementList.push(minusOneReplacement)
11:  replacementList.push(plusOneReplacement)

```

4.3.3.4 Operator Mutations

The Operator Mutation consists of generating one mutant for each other operator of the same class as the original one, as demonstrated in Algorithm 4. Operator mutation is a mutation that generates a lot of mutations, and in future versions, limiting the possible mutations might be considered. Types of operators included in our tool revolve around:

- **Arithmetic Operators** - $/, \%, *, -, +$;
- **Comparison Operators** - $<, <=, ==, ===, !==, !=, >=, >$;
- **Logical Operators** - $\&\&, ||$;
- **Update Operator** - $++, --$.

Operator Mutations are applied to every **binary expression**.

Algorithm 4: Generate Operator Mutants

```

1: procedure GenerateOperatorVariants(node, replacementList)
2:   operator  $\leftarrow$  node.op
3:   for operatorType in operatorsTypeList do
4:     if operator.includes(operatorType) then
5:       for newOperator in operatorType do
6:         newNode  $\leftarrow$  new BinaryExp(node.lhs, newOperator, node.rhs)
7:         replacement  $\leftarrow$  new Replacement(node, newNode)
8:         replacementList.push(replacement)
9:       end
10:    end
11: end

```

4.3.3.5 Boolean Mutations

The Boolean Mutation consists of generating one mutant for each **boolean** found, inverting its value, as demonstrated in Algorithm 4 (p. 36).

Algorithm 5: Generate Boolean Mutants

```

1: procedure GenerateBooleanVariant(node, replacementList)
2: newNode  $\leftarrow$   $!(node == true)$ 
3: replacement  $\leftarrow$  new Replacement(node, newNode)
4: replacementList.push(replacement)

```

4.3.3.6 Remove Prefix Mutations

Similar to boolean mutations, Remove Prefix Mutations intend to invert the value of a given expression, in this case, by removing a **!** prefix, as demonstrated by Algorithm 6.

Remove Prefix Mutations are applied to every **prefix unary expressions**.

Algorithm 6: Generate Remove Prefix Mutants

```

1: procedure GenerateRemovePrefixVariant(node, replacementList)
2: newNode  $\leftarrow$  ' '
3: replacement  $\leftarrow$  new Replacement(node, newNode)
4: replacementList.push(replacement)

```

4.3.3.7 Statement Moving Mutations

The Statement Moving Mutation, as demonstrated in Algorithm 7, is fundamentally different from the previous mutations. Instead of manipulating the AST, it act upon the source code, by switching each pair of adjacent lines between themselves.

Algorithm 7: Generate Statement Moving Mutants

```

1: procedure GenerateStatementMovingVariants(code, replacementList)
2: linesList  $\leftarrow$  code.getLines()
3: for i  $\leftarrow$  0 to linesList.length - 1 do
4:   newCode  $\leftarrow$  SWITCHLINES(code, i, i + 1)
5:   replacement  $\leftarrow$  new Replacement(code, newCode)
6:   replacementList.push(replacement)
7: end

```

4.3.4 pAPRika Extension

As the mutant generation process is rather resource-intensive, running our tool on every code change would be both ineffective, inefficient and unnecessary. Therefore, our tool runs on

specific events, which can be deactivated/activated in the extension's settings - through the `workspace/didChangeConfiguration` notification - enabling higher control of the extension by the user (cf. Figure 4.4).

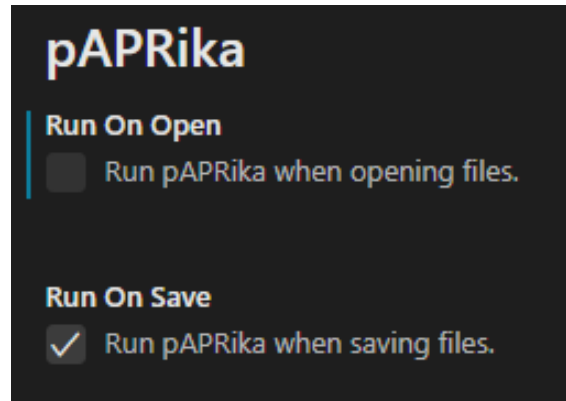


Figure 4.4: Tool's settings within the settings menu of Visual Studio Code.

Our extension can only be activated in supported languages, *i.e.* JavaScript and TypeScript, and runs on:

- **Open** - Whenever a new document is opened - through the `textDocument/didOpen` notification.
- **Save** - Whenever a new document is saved - through the `textDocument/didOpen` notification.
- **Command** - Whenever the user uses one of two IDE commands, *pAPRika: Spice this file* or *pAPRika: Spice all open files* - through the `workspace/executeCommand` notification.

Additionally, while the tool is running, the language server sends a progress notification, through `window/workDoneProgress/create`, which allows developers to know the state of the extension (cf. Figure 4.5).

Regarding the process itself, after the extension is run, the tool is composed of the next elements, which are essential in the interaction with the developer and to provide the intended live feedback.



Figure 4.5: Display of the tool's progress on the bottom left corner of Visual Studio Code.

4.3.4.1 Test Suite Requirements

Since mutations are not applied randomly, and need an AST as a basis for applying said mutations, the function being tested needs to be identifiable in the tests that compose a test suite. On Diogo Campos' implementation, tests had to identify the tested function with `#fix functionName`, where `functionName` was, as the name implies, the name of the function being tested. Such

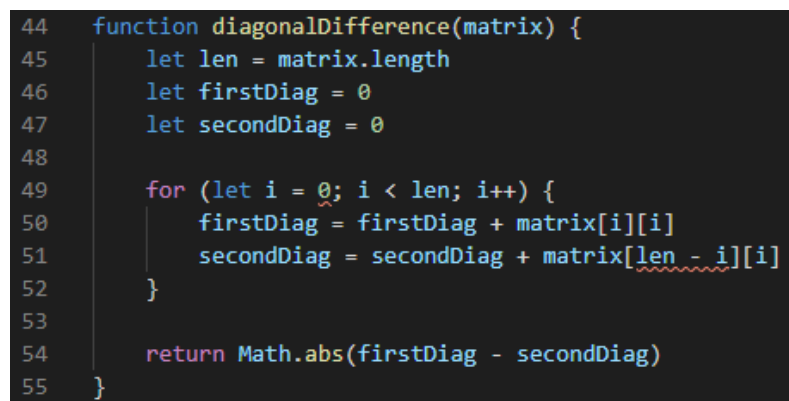
implementation mostly worked, however, as soon as the function was not declared in the "basic" function declaration it would not be caught by the function AST seeking algorithm.

Our extension now has support for all function declarations as per Mozilla's JavaScript Functions Reference ⁸, which included arrow functions, method declarations, function expressions, and property definitions, however, method declarations and property definitions require an additional identification component. In short, pAPRika requires a less wordy identifier, `className.functionName`, to be inserted in each of the tests to be able to identify the AST under test, where `className.` is only required for method declarations and property definitions.

One of the most important components in our extension is the *mocha* feature-rich JavaScript and TypeScript test framework⁹. *Mocha* is one of the most used test-driven development frameworks within Node.js¹⁰, though, our use of this framework will not be in its most common form. Our solution takes advantage of Mocha's programmatic API¹¹, which enables us to run tests for both the base source code, as well as the mutant files generated by the tool.

4.3.4.2 Display Suggestion

As we wanted to reduce the validation period of general automatic program repair techniques, we needed to delve into the immediacy of suggestions, presenting suggestions while the developer is still with the thought process aligned with the function being tested, instead of after, reducing the required effort in inspecting software repair techniques' output. Therefore, the tool's ease of use and access was of the utmost importance, which meant that bug locations had to be well highlighted. For this reason, we opted for demonstrating potential fix locations as if they were semantic suggestions. These suggestions were sent to Visual Studio Code as a Diagnostic ¹² - through the `textDocument/publishDiagnostics` notification - as per their API.



```

44  function diagonalDifference(matrix) {
45      let len = matrix.length
46      let firstDiag = 0
47      let secondDiag = 0
48
49      for (let i = 0; i < len; i++) {
50          firstDiag = firstDiag + matrix[i][i]
51          secondDiag = secondDiag + matrix[len - i][i]
52      }
53
54      return Math.abs(firstDiag - secondDiag)
55  }

```

Figure 4.6: Underlined potential fix locations by pAPRika.

⁸Mozilla's JavaScript Reference - Functions (Retrieved by: 2 July 2020)

⁹Mocha (Retrieved by: 20 June 2020)

¹⁰Node.js (Retrieved by: 20 June 2020)

¹¹Mocha - Programmatic API (Retrieved by: 20 June 2020)

¹²Visual Studio Code's API - Diagnostic (Retrieved by: 1 July 2020)

We believe this is the best possible implementation of the suggestion display, as after the Quick Fix (cf. Section 4.3.4.3) is applied, we are confident that it passes all tests defined in the test suite, and, therefore, should at least be analysed by the developer.

As one can observe, our implementation (cf. Figure 4.6, p. 39) already represents a big improvement over Diogo Campos' implementation (cf. Figure 4.7), as more than one diagnostic can be displayed per file. For the used example, such might not have a big impact, as all potential fixes result from the same loop, and offer very similar suggestions. However, it was chosen for practicality as we can better represent and compare, in a small example, the resulting difference.

```

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 4.7: Underlined potential fix location in Diogo Campos' tool [Cam19].

All fix suggestions produced by our tool appear in the problems tab of the IDE for easy visualisation of all problems at once (cf. Figure 4.8), which we hope helps the developer assess which fix might be the best, as they are the best final gatekeeper for current APR tools [MBC⁺19], including our own, and more information leads to better and conscious decisions.

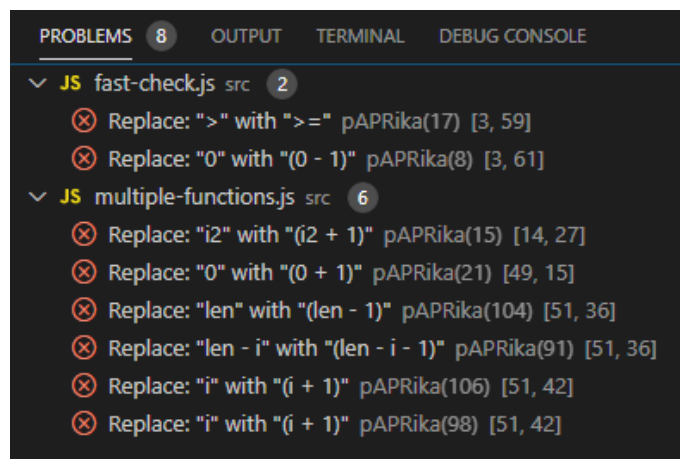
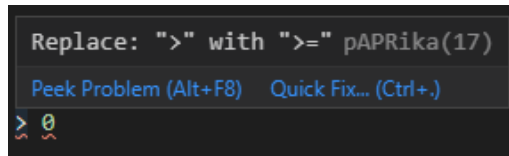


Figure 4.8: List of problems generated by the tool.

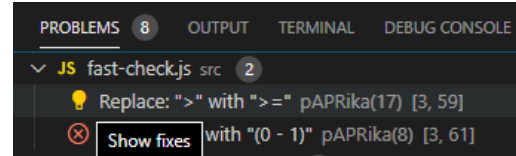
4.3.4.3 Quick Fix

After displaying to the user the potential fix locations, if the developer opts for applying the fix, as suggested by the tool, two ways were implemented of accepting the suggested fix, either

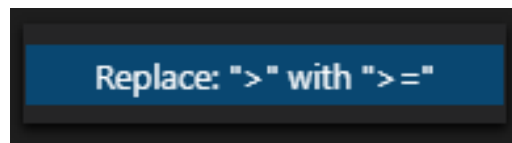
by using a Quick Fix by hovering over the underlined suggested fix location (*cf.* Figure 4.9a) or by hovering one of the suggestions in the problems list (*cf.* Figure 4.9b), as both send a `textDocument/codeAction` notification to the language server, as per Visual Studio Code's API `CodeAction`¹³.



(a) Fix suggestion presented on hover of a suggested fix location.



(b) Fix suggestion presented on hover of a specific problem within the IDE's list of problems.



(c) Replacement message after quick fix selection.

Figure 4.9: Deployment of fix suggestions.

After the developer clicks the *Quick Fix* button (also represented as the light bulb), in either of locations, a small window shows up where the developer has to click again to confirm the change as shown in Figure 4.9c.

After the change is confirmed, the code is replaced and the diagnostics are cleared.

4.3.5 Property-Based Testing Framework

One of the most important objectives of our proposed solution was to implement property-based testing support within our tool. Properties are mere statements that can be defined on paper, but without a property-based testing framework, testing such properties would be an impossible task. PBT is a technique widely used in the Haskell community thanks to QuickCheck [CH00]. The idea consists in automatically generating inputs for testing a function. Frameworks such as QuickCheck merely provide an environment where we can verify the truthfulness of said properties. With this objective in mind, we set out to research the most optimal JavaScript/TypeScript frameworks for our purpose.

Our research concluded that, to this day, there are 5 libraries that are capable of offering primitives to perform property-based testing:

- *fast-check*¹⁴, developed by Nicolas Dubien since 2017
- *jsverify*¹⁵, developed by Oleg Grenrus since 2014
- *testcheck*¹⁶, developed by Lee Byron since 2014

¹³Visual Studio Code's API - `CodeAction` (Retrieved by: 2 July 2020)

¹⁴*fast-check* (Retrieved by: 20 May 2020)

¹⁵*jsverify* (Retrieved by: 20 May 2020)

¹⁶*testcheck* (Retrieved by: 20 May 2020)

- *jscheck*¹⁷, developed by Douglas Crockford since 2013
- *quick_check.js*¹⁸, developed by Jakub Hampl since 2014
- *proptest*¹⁹, developed by Dan Rosén in 2018

All of these frameworks served their purpose of providing a way of providing access to the generative testing that PBT provides. However, some of these frameworks have completely lost all developer support, and are somewhat deprecated, as noted by *quick_check.js*'s creator who suggests new users to use *fast-check*, and as we can observe from *npm trends*²⁰ library popularity.

Thus, for further testing only *fast-check*, *jsverify* and *testcheck* were considered. However, we quickly noticed that all of these three frameworks support mocha integration, and therefore, are supported by our tool. Though, It is important to note that *jsverify*'s maintainer has openly abandoned further work²¹, and *testcheck* has their development stalled, and *fast-check* is the only framework that is still being updated, with numerous articles indicating that better results can be obtained with it²².

4.3.6 TypeScript Support

TypeScript's utilization among the developer community has been growing steadily for the past years, thus, TypeScript support was defined as one of the main objectives for this tool. Although previous implementations of this extension already took advantage of TypeScript's Compiler API, it lacked true TypeScript support. As such, in order to transpile our `*.ts` files to pure JavaScript for Node.js, and its engine, to understand them, we used *ts-node*. Such implementation, however, like any TypeScript project, requires a *tsconfig.json* file to indicate any necessary details to the transpiler.

4.3.7 General Improvements

Throughout the development of this tool a subconscious goal was also defined, to open-source pAPRika, and to that extent a lot was done to improve the overall quality of the project besides the features listed above.

First and foremost, there was a major code refactoring, accompanying the adaptation to the language server protocol, with resulted in improved code readability, elimination of hard-coded variables, improved error handling, and the creation of understandable documentation.

Secondly, the extension, which was only locally buildable, became fully publishable with the Visual Studio Code Extension Manager, and integrated within a continuous integration/continuous delivery process, opening the doors towards a truly maintainable open-source project.

¹⁷*jscheck* (Retrieved by: 20 May 2020)

¹⁸*quick_check.js* (Retrieved by: 20 May 2020)

¹⁹*proptest* (Retrieved by: 20 May 2020)

²⁰*npm trends - fast-check vs jsverify vs testcheck vs jscheck.js vs proptest vs quick_check* (Retrieved by: 20 May 2020)

²¹*jsverify's Issues - Looking for a new maintainer* (Retrieved by: 21 May 2020)

²²*Property-Based Testing in JavaScript* (Retrieved by: 20 May 2020)

Finally, the whole repository suffered an overhaul, with the creation of a complete *README* file²³ and of a website to advertise the tool and serve as an entry point towards new users²⁴.

4.4 Summary

This chapter described the proposed solution and implementation in great detail, by initially providing a brief contextualization of the implemented solution and clearly defining its objectives.

Subsequently, we described every aspect of our implementation, in regards to the APR technique, the use of the language server protocol, mutation generation, the extension itself, PBT framework integration and TypeScript support.

Tufano et al. state that the two major problems of automated repair approaches are the ability to produce fixes that are acceptable to the programmer and overfitting to test cases [TPW⁺18]. It was also found by *Qi et al.* that most generate and validate techniques are not correct, and achieve their repairs through the deletion of important functionality. Taking this into account, our solution does not attempt to apply fixes without a developer, as it is a mere tool that intends to help the developer's reasoning. Additionally, our tool supports PBT, which we believe eliminates the overfitting problem, and does not attempt to be too greedy with its mutations, instead, it opts to find and repair the small bugs that developers, a lot of the times, fail to discover.

²³Make a README (Retrieved by: 15 April 2020)

²⁴pAPRika Website (Retrieved by: 30 June 2020)

Chapter 5

Empirical Evaluation

5.1	Objectives	45
5.2	Guidelines	46
5.3	Planning	47
5.4	Tasks	50
5.5	Results	55
5.6	Threats to Validity	69
5.7	Discussion	71
5.8	Summary	75

The purpose of this dissertation was to validate whether or not we could diminish the debugging time of developers, by introducing a program repair process as an IDE extension, *i.e.* within the development phase instead of after. Such would combine the advantages of both the developer’s reasoning and the machine-generated repairs, as repairs would be merely suggestions to improve the developer’s thought process, resulting in a more streamlined cross-validation process. In order to validate the performance of the tool relative to the hypothesis and research questions (*cf.* Section 3.3, p. 22), we expanded a previously developed controlled experiment [Cam19] and envisioned variations for it, incorporating the new features and capabilities of the tool. Thus, in this chapter, we detail the empirical evaluation conducted, throughout its objectives, (*cf.* Section 5.1), guidelines (*cf.* Section 5.2, p. 46) and planning (*cf.* Section 5.3, p. 47), tasks performed, (*cf.* Section 5.4, p. 50) and, consequently, the respective results obtained and their extensive analysis in Section 5.5 (p. 55), and also the main validation’s threats (*cf.* Section 5.6, p. 69). Finally, we present a brief discussion of all obtained results (*cf.* Section 5.7, p. 71).

5.1 Objectives

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.

Science is based on experimental validation and repetition, allowing us to corroborate hypotheses, assumptions, and conjectures, and it has been proven again and again that such is the case of Software Engineering [FPG94]. Taking this into account, a controlled experiment was envisioned to validate the hypothesis and research questions presented in Section 3.3 (p. 22). Throughout this controlled experiment, our objective was always to extend the experimental validation that was previously carried out by Campos [Cam19] with our updated tool, that is more feature-complete, solid and capable, as shown in Section 4.3 (p. 27). With this in mind, we set to recreate the already developed tasks, while also expanding and integrating new tasks with properties, as defined in Section 5.4.1 (p. 50). Section 5.4.2 (p. 52) goal is to evaluate how well our solution worked when tested by developers, namely, whether it reduced debugging time, by helping with their solution converge towards an optimal solution, therefore, reducing development time, and improved code quality. To collect such metrics, we set out to evaluate developer performance when using the implemented extension, which even though it can be easily extensible to other IDEs, we chose to only use one IDE, Visual Studio Code, both to reduce the number of variables in our study and reduce implementation time. Thereafter, we compare their performance to users who did not use such APR tool, by measuring the time to solution for each problem, while also assessing the quality, form, and naturalness of the final code developed.

In light of such objectives, we set out to split all participants into two questionnaires. Firstly, we will talk about, what we will refer to as, the general public questionnaire (*cf.* Appendix A, p. 81), whose objective was to evaluate the participants' view and understanding of properties, and whether they consider them useful. We also analyse the participants' response to PBT frameworks and their view on the ease of adoption of property-based testing in their testing workflow. Secondly, the usability questionnaire (*cf.* Appendix B, p. 89) will, in addition to the general public questionnaire's content, evaluate the use of the developed tool. While using our benchmark, it was expected that the experiment participants who used the tool had higher performance, however, we wanted to measure the degree of improvement and whether or not their solutions were understood by them. Furthermore, as a means of verification, we tried to evaluate if the participants who used the tool could understand its purpose, its uses, its functionalities, and its limitations.

Thus, the two main purposes of this study are to obtain results that allow a more direct comparison between pAPRika users *versus* users that solely use normal test-driven development with example-based test suites; and to obtain reactions from the participants to property-based testing as a component in their development workflow. These will allow us to validate the hypothesis presented in Section 3.3 (p. 22) as a potential solution to the problems previously described in Section 3.2 (p. 20), and to also answer the enumerated research questions.

5.2 Guidelines

To properly validate the use of the developed tool contemplated in Chapter 4 (p. 25), one must define action lines to narrow the scope of the validation and properly achieve the objectives of this empirical study and its main hypothesis. We will start by enumerating our guidelines for our

experiment:

Participants - Since this tool is built for developers, it is of the outmost importance that all participants have the knowledge to comprehend and analyse software. Thus, all participants should have at least some experience in the language being tested, JavaScript, and be in the computer engineering field, in particular the area of software development.

Expertise - Since testing frameworks, especially in JavaScript, are not that common in Computer Engineering degrees, while its presence in the real world is well-established, we hope to nourish some knowledge of their use and the benefits of Test-Driven Development.

Participant motivation - Participants should consent that their participation is a fruit of their own will, and that all their opinions will be honest and critical before partaking in this experiment, where they will experience a different approach to debugging.

Duration - Overall, the experiment's duration is not limited, however, in tasks from the usability questionnaire, there is a maximum duration for each task, as better clarified in Section 5.3.2 (p. 48).

Reliability - For the results of this experiment to be reliable and trustworthy, participants shall not have had any previous experience with the tool being tested. Additionally, due to limitations imposed by the global pandemic, all questionnaires were filled where the participants deemed desirable, though participants of the usability questionnaire were closely monitored, as it required monitoring from the overseer. The specific data integrity details of both questionnaires are further described in Section 5.3.

5.3 Planning

Pushing developers towards a faster and more interactive debugging experience while programming has been one of the major objectives of the development of this tool, therefore, the performance of a tool like this has to be well measured. Thus, one must evaluate whether it truly improves the developer's debugging experience and allows for a faster convergence to a solution. Additionally, as our objective for this dissertation was also to analyse the possibility of using property-based tests as specifications, we also examine the participant's ability to come up with example-based tests, understanding of properties and PBT frameworks, and the general opinion of a tool such as pAPRika after a brief video demonstration.

Hence, our controlled experience requires careful prior planning to identify and establish its rules, steps and measurements, which we will proceed to enumerate below. However, since evaluating the usability is a time-consuming undertaking in and of itself, we decided to create two questionnaires, the General Public Questionnaire (*cf.* Section 5.3.1) and the Usability Questionnaire (*cf.* Section 5.3.2, p. 48), where the only difference between them is that the latter has the additional usability section.

5.3.1 General Public Questionnaire

Participants - All participants should participate in this experience voluntarily and spontaneously, and should be genuinely interested in the use of a tool such as the one developed. As mentioned in

the chapter above, direct contact with computer engineering and programming is a requirement, and software development experience and a minimum of basic familiarity with JavaScript is also required, as the tests will consist of repairing faulty JavaScript code. Participants for this questionnaire will all be from various curricular years of the Integrated Master in Informatics and Computing Engineering.

Tool exposure - Participants have all shown interest in participating in this experiment, as they, in a broad sense, considered that such a novelty tool that they had never had experience with, with the described features, could be very useful to them in their coding processes, assisting in their software development.

Questionnaire - We developed this base questionnaire to help answer some different goals of the experiment. Namely, the users' background, assessing their technical knowledge and familiarity with the language and testing framework used, ability to come up with example-based tests, understanding of properties and PBT frameworks, and a post-test survey evaluating the usability, performance, and other factors of the tool. Statements about the users' background were registered using a Likert scale [Lik32] with five possible responses: Strongly Disagree, Somewhat Disagree, Neutral, Somewhat Agree and Strongly Agree. The remaining questions presented to participants were multiple-choice questions pertaining to their ability to come up with example-based tests and understanding of properties and PBT frameworks.

Duration - After the creation of this questionnaire, we estimated that participants should take around 5 minutes to complete the questionnaire. Participants were previously informed of this duration estimation.

Data integrity - As previously stated (*cf.* Section 5.2, p. 46), participants were required to give their consent that their opinions were honest and critical, and that their participation was of their own will.

5.3.2 Usability Questionnaire

As mentioned at the beginning of this Section 5.3, the only difference of the General Public Questionnaire to the Usability Questionnaire is the additional usability section of the questionnaire. As such, in this section, we will only detail additional factors resultant from the usability portion of the Usability Questionnaire, and potential differences to Section 5.3.1 (p. 47). The usability section of this questionnaire is replicated from Campos work [Cam19], as such, the results are expected to be similar, to better assert our hypothesis.

Participants - Participants for this questionnaire, unlike the general public questionnaire, were solely from the last year of the Integrated Master in Informatics and Computing Engineering.

Tool exposure - All participants were briefly introduced to the tool with an example exercise before their set of tasks with the tool.

Questionnaire - Additionally to the questions present in the general public questionnaire (*cf.* Section 5.3.1, p. 47), this questionnaire presented participants with statements about their experience and performance after each of the parts in the study. All statements in this additional

section were also registered using a Likert scale [Lik32], as it was used when surveying participant's background.

Duration - As the set of tasks introduced by this questionnaire, in comparison to the general public questionnaire, is expected to last longer, we, firstly, estimated the duration of these tasks, to ensure participants could allocate the required time without interruptions. It was then estimated that the tasks should take around 25 minutes, which, when aggregated with the duration of the questions in the general public questionnaire, should equate to around 30 minutes. However, 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 timeout for each task of 7 minutes and 30 seconds, motivating volunteers to optimise their time management.

Environment - To ensure that everyone was accustomed to their operating system, keyboard, mouse, etc. everyone participated in this controlled environment through their computer. This also ensured that pandemic confinement rules were met during the study, without endangering participants. Everyone used Visual Studio Code since the current version of the tool only supports it, even though it is expandable to other IDEs, and participants were also authorised to access the Internet for research purposes while solving the tasks. Since the environment was not the same for everyone, a small script was created to guide participants in the installation of the extension and download of the benchmarks.

1. Fill the questionnaire until you are asked for a group.
2. Download the benchmark from the provided link, extract it and open the folder in Visual Studio Code.
3. Download and install the tool. The extension is provided as a VSCode extension executable, therefore the installation has to be through the Visual Studio Code's Extensions menu, selecting "More actions..." represented by "...", choosing "Install from VSIX...", and, finally, selecting the provided VSCode extension executable.

Afterwards, after a brief explanation of the tool's inner workings and functionalities, the overseer provided the participant with the group identification, and granted the green light to proceed with the questionnaire.

Difficulty - The tasks' difficulty is considered to be easy as the bugs to be found on the several tasks is always solvable by one or two changes, however, the experiment's overseer has, throughout the experience, tried to understand the difficulties of each of the participants, while additionally providing a question in the survey to ensure that the overseer's perception was correct.

Data integrity - In addition to the consent users gave, as per Section 5.3.2 (p. 48), during the experiment, both the time to reach a solution and the quality of the code solutions were measured and evaluated, in addition to whether participants used the tool when the task allowed them to or not.

Procedure - To compare developer performance while using the pAPRika 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 there were no set guarantees of *a priori* technical knowledge parity, a need for both groups to be subjected to the extension has led to the creation of two different, but equivalent, problem sets. The usability portion of the questionnaire was therefore comprised of two different parts, in accordance with the following procedure:

The first part of the usability portion required:

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

The second part of the usability portion required:

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

The tasks used in each of the problem sets is available in Section 5.4.2 (p. 52).

5.4 Tasks

As described in Section 5.3 (p. 47), we have decided to create two questionnaires, the General Public Questionnaire (*cf.* Section 5.3.1, p. 47) and the Usability Questionnaire (*cf.* Section 5.3.2, p. 48), therefore, the Section 5.4.1 will pertain to both questionnaires, and Section 5.4.2 (p. 52) will only refer to the usability portion of the questionnaire.

5.4.1 General Public Tasks

With property-based testing frameworks rising in popularity due to their automation of what would otherwise be tedious manual labour, in this questionnaire, we set out to validate our hypothesis, which states that with the use of properties as specifications we can eliminate overfitting in APR fixes, and to answer our research questions (*cf.* Section 3.3, p. 22).

With that in mind, we have created three different problems, with two cases each, to assess the participants' ability to create complete example-based tests, to formulate properties, and to create property-based tests using existing frameworks, respectively. The two distinct cases in which the three problems were evaluated were: *myParseInt*, a re-implementation of JavaScript's `parseInt`, and an implementation of the longest common substring algorithm.

Task 1

In the first task, we present participants with example-based tests of both `myParseInt` (*cf.* Figure 5.1, p. 51) and `longestCommonSubstring` (*cf.* Figure 5.2, p. 51).

Based on the given examples, participants were asked, for both examples, if the tests provided were enough to test the function in question. This question is multiple-answer with the following possible answers:

- “Yes, they seem to cover most of the spectrum of possibilities.”
- “No, but I CAN’T think of more different test cases.”
- “No, and I CAN think of more tests to incorporate.”

```
describe('myParseInt', function () {
  it("should return same values as JavaScript's parseInt(). #fix {myParseInt} (1)", function () {
    expect(myParseInt('10')).toEqual(parseInt('10'))
  })
  it("should return same values as JavaScript's parseInt(). #fix {myParseInt} (2)", function () {
    expect(myParseInt('00 abs')).toEqual(parseInt('00 abs'))
  })
  it('should return NaN. #fix {myParseInt} (3)', function () {
    expect(myParseInt('abc 95')).toBe.NaN
  })
  it('should return NaN. #fix {myParseInt} (4)', function () {
    expect(myParseInt(' asd45.23')).toBe.NaN
  })
  it("should return same values as JavaScript's parseInt(). #fix {myParseInt} (5)", function () {
    expect(myParseInt('999.4')).toEqual(parseInt('999.4'))
  })
})
```

Figure 5.1: Sample of example-Based tests for myParseInt.

```
describe('Longest Common Substring', () => {
  it('should find "sentence" {longestCommonSubstring}', () => {
    assert.equal(
      longestCommonSubstring(
        'This is a sentence that will get analysed a lot',
        'There are a lot of sentences with a lot of common substrings'
      ),
      'sentence'
    )
  })

  it('should find "mportant" {longestCommonSubstring}', () => {
    assert.equal(longestCommonSubstring('Wording is important', 'Important wording'), 'mportant')
  })

  it('should find "ika" {longestCommonSubstring}', () => {
    assert.equal(longestCommonSubstring('pAPRika', 'paprika'), 'ika')
  })
})
```

Figure 5.2: Sample of example-Based tests for longestCommonSubstring.

Task 2

In the second task, we present participants with a brief description of what a property is, and with properties of both myParseInt (cf. Figure 5.3, p. 52) and longestCommonSubstring (cf. Figure 5.4, p. 52).

Based on the given examples, participants were asked, for both examples, if they consider that they could have come up with the given properties. This question is multiple-answer with the following possible answers:

- “Yes, and I CAN think of a few more.”
- “Yes, but I CAN’T think of more.”
- “No, they seem hard to come up with.”

Task 3

```

for all (x: integer)
where y is the stringified version of x
myParseInt(y) should be equal to parseInt(y)

for all (x: double)
where y is the stringified version of x
myParseInt(y) should be equal to parseInt(y)

```

Figure 5.3: Example properties for `myParseInt`.

```

for all (x, y)
longestCommonSubstring(x, y) should be equal to
    longestCommonSubstring(y, x)

for all (x, y)
the result of longestCommonSubstring(x, y) should be present
    in both x and y

for all (x, y, z)
the result of longestCommonSubstring(y + x + z, x) should be x

```

Figure 5.4: Example properties for `longestCommonSubstring`.

In the third task, we present participants with an implementation of the properties given in task 2 in a property-based testing framework of both `myParseInt` (cf. Figure 5.5, p. 53) and `longestCommonSubstring` (cf. Figure 5.6, p. 53).

Based on the given examples, participants were asked, assuming that the example-based tests were incomplete and that property-based testing would widen the spectrum of test possibilities, for both examples, if they would consider writing property-based tests. This question is multiple-answer with the following possible answers:

- “Yes, and they seem simpler than having to think of all the test cases by myself.”
- “Yes, even if the framework seems a bit complicated at first.”
- “No, I would prefer to think of the test cases myself.”

5.4.2 Usability Tasks

Benchmarks are crucial to high-quality empirical science, so, choosing which one to use is a key step in towards validation. There are plenty of automated program repair benchmarks¹, however, these tend to assess repair quality of the implemented tools, instead of developer usage. In light of such absence, Diogo Campos [Cam19] tailored a dataset specifically for this study while taking into account its objectives and scope, which we believe was well built and objective.

¹Program-Repair.org (Retrieved by: 28 June 2020)


```
describe('Properties of myParseInt', function () {
  it('parsing integers {myParseInt}', () => {
    fc.assert(
      fc.property(fc.integer(), function (num) {
        const string1 = fc.stringify(num)
        expect(myParseInt(string1)).to.equal(parseInt(string1))
      })
    )
  })

  it('parsing doubles {myParseInt}', () => {
    fc.assert(
      fc.property(fc.double(), function (num) {
        const string1 = fc.stringify(num)
        expect(myParseInt(string1)).to.equal(parseInt(string1))
      })
    )
  })
})
```

Figure 5.5: Sample of example-Based tests for myParseInt.

```
describe('Properties of Longest Common Substring', () => {
  it('should find the same substring lengths whatever the order of the inputs {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), (s1, s2) => {
        assert.equal(longestCommonSubstring(s1, s2).length, longestCommonSubstring(s2, s1).length)
      })
    )
  })

  it('should include the substr in both strings {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), (s1, s2) => {
        const longest = longestCommonSubstring(s1, s2)
        assert.ok(s1.includes(longest))
        assert.ok(s2.includes(longest))
      })
    )
  })

  it('should detect the longest common {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), fc.string(), (s, prefix, suffix) => {
        assert.equal(longestCommonSubstring(prefix + s + suffix, s), s)
      })
    )
  })
})
```

Figure 5.6: Sample of example-Based tests for longestCommonSubstring.

Regarding this work, as described in Section 4.3.1, *one unit* is defined as *one function*, as a result, each problem will correspond to a function. To evaluate the usability of our extension we have set three unique types of problem, according to whether or not, and when the extension would find a solution:

1. **Immediate:** Functions whereat a single bug is present and visible, and the extension is immediately capable of finding a solution for it.
2. **Nonimmediate:** Functions whereat a single bug is present and visible, but the extension is only capable of finding a solution for it after the addition of a piece of code, analogous to missing functionality.
3. **Nonpresent:** Functions whereat a single bug is not present nor visible since participants still

need to write a piece of code, analogous to missing functionality, which may or may not contain a bug that the extension is able to fix.

Immediate problems sole purpose is to be used as proofs of concept, demonstrating the effectiveness of the tool for simple challenges, while also serving as sanity checks. Whereas nonimmediate and nonpresent functions serve as an evaluation point of the tool in solving more complex problems, being used to perceive the effectiveness of the tool with bugs not introduced by the developer and with bugs introduced by the developer, respectively.

Since we are creating a controlled experiment, we must clarify our process for identifying the generated mutants as potential fixes. Therefore, mutants must only be classified as fixes if they fulfil all the predefined criteria (*cf.* Section 4.3.1, p. 28) for a valid suggestion. As we believe that overfitting can be an issue of example-based tests in APR techniques, the following problems as defined by Campos [Cam19], which were also used in this study, were carefully selected to only present users with valid suggestions. Since our objective was never to evaluate the tool's suggestion quality, as we believe such derives from the quality of the test suite, we considered tasks as solved as soon as they passed the predefined unit tests. As such, and according to the procedure defined in Section 5.3.2, the following problems were used:

Task 1

Starting with the first task, we present participants with an **immediate** problem. This task only requires an operator change for example, allowing us to validate the extension in terms of its basic usability.

In **Problem Set X**, a fault was introduced in one implementation of Bubble Sort, by inverting a comparison operator.

In **Problem Set Y**, a problem from the */r/dailyprogrammer subreddit*² was used, for which we present an adapted function, of the JavaScript solutions proposed in the comments section, where its goal is to find if every letter appearing in the input string does so the same number of times.

Task 2

We then present participants, in the second task, with another **immediate** problem. This task only requires an off-by-one operator change for example, allowing us to continue validating the extension in terms of its basic usability, and serving as a sanity check.

In **Problem Set X**, a popular question used in various programming challenges that, given a matrix, requires the absolute difference between the two diagonals, was adapted.

In **Problem Set Y**, a problem from the */r/dailyprogrammer subreddit*³ was used, for which we present an adapted function, of the JavaScript solutions proposed in the comments section, where its goal is to calculate the score of a game based on an input string.

Task 3

Thirdly, we present participants with a **nonimmediate** problem, where we intend to assess the behaviour of the extension in circumstances in which the bug is present but is not found

²[2019-01-14] Challenge #372 [Easy] Perfectly balanced (Retrieved by: 28 June 2019)

³[2018-05-14] Challenge #361 [Easy] Tally Program (Retrieved by: 28 June 2019)

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.

In **Problem Set X**, a problem from the */r/dailyprogrammer subreddit*⁴ was used, for which we present an adapted function, of the JavaScript solutions proposed in the comments section (with a removed line of code and an inverted operator), where its goal is to calculate 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 .

In **Problem Set Y**, a problem from the */r/dailyprogrammer subreddit*⁵ was used, for which we present an adapted function, of the JavaScript solutions proposed in the comments section (with a removed line of code and an inverted operator), where its goal is to, given the amount to give and a list of coins, calculates the minimum number of coins required to reach that amount.

Task 4

The fourth and final task, a **nonpresent** problem, had the goal of evaluating the extension when the bug is introduced by the developer. Neither problems had a bug previously introduced, however, the functionalities that must be implemented are highly prone to bugs.

In **Problem Set X**, the goal was for participants to implement a function that returned a substring between two indexes. However, such indexes are inclusive, unlike JavaScript's `string.substring` implementation.

In **Problem Set Y**, the goal was equivalent to, where participants had to implement a function that returned a slice of an array between two elements. Again such indexes are inclusive, unlike JavaScript's `string.slice` implementation.

5.5 Results

This section will now present and analyse the results from both of the controlled experiments, *i.e.* the usability questionnaire and the general public questionnaire, along with a brief analysis for each task and section of the questionnaires.

Regarding the statistical methods utilised, for the sake of consistency, Likert-type [Lik32, Alb97] scales were used for both the background, and the tool usability evaluation, since, throughout this controlled experiment, our objective was always to extend the experimental validation that was previously carried out by Campos [Cam19]. Conversely, there were some multiple-choice and multiple-response questions that were analyzed through charts.

First and foremost, we will describe some of the common sections of both questionnaires, *i.e.* participants' profile (*cf.* Section 5.5.1, p. 56) and background (*cf.* Section 5.5.2, p. 56), and, afterwards, analyse its results in Sections 5.5.4 and 5.5.5.

⁴[2016-10-03] Challenge #286 [Easy] Reverse Factorial (Retrieved by: 28 June 2019)

⁵[2018-01-29] Challenge #349 [Easy] Change Calculator (Retrieved by: 28 June 2019)

5.5.1 Participants' Profile

For this controlled experiment, gender or age were considered characteristics that have no significance. On the other hand, academic qualifications and years of professional experience might have a higher impact as they are important factors to analyze, since many of the participants are still enrolled in an academic degree, and, therefore, technological knowledge might vary, among the several stages of a degree. Furthermore, the number of years of professional experience is important as TDD is widely more popular in the professional environment, when compared to the academic environment.

5.5.2 Background

The background section of the questionnaires, consists of multiple Likert-type items, forming a Likert-Type scale, that will assess the participants' comfort level with the languages, frameworks and technologies used in this experiment. This scale is comprised of the eight questions, 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.

Participants were then given a score based on their answers, where, on a scale of -2 (Strongly Disagree) to 2 (Strongly Agree), answers were summed, resulting in the final score. Intrinsically, this equates to a minimum possible score of -16, a maximum score of 16, and an average of 0. However, we expect our average to be overall higher due to the participants' academic qualifications.

To better understand whether or not there are significant statistical differences between both groups, we will test the null hypothesis that both groups have identical means. For this test, like Campos in his work [Cam19], we have used the *Student's t-test*, a test that relies on the assumption of a normal distribution of the population and equal variances. Though, multiple studies have shown it is fairly robust to violations of at least one assumption [KHL11], and capable of handling minor sample sizes [dW13].

However, before calculating the *Student's t-test*, we will calculate *Levene's test* for "equality of variances, which tests the null hypothesis that both groups are from populations with equal variances" [BF74].

Secondly, the *Shapiro-Wilk test* will be used, which tests both groups against the null hypothesis that "the sample comes from a population with a normal distribution" [SW65].

If, with both of these tests, we may not reject the null hypothesis, we will then can assume that

both samples come from normally distributed populations, and that their variances are equal. From here, we can proceed to use the *Student's t-test* to test for the following hypotheses:

H0 (Null Hypothesis): *The means of the populations from which each group was sampled are equal.*

H1 (Alternate Hypothesis): *The means of the populations from which each group was sampled are different.*

5.5.3 Post-Test Survey

As a final section of both questionnaires, participants will be asked to provide with small feedback of the tool developed in this dissertation.

Firstly, this section consists of multiple Likert-type items, forming a Likert-Type scale, that will assess the participants' comfort level with the tool and its features:

- **F1** - The tool's features were simple to use and easy to understand.
- **F2** - This tool can positively impact my development workflow.
- **F3** - I would consider using this tool.
- **F4** - A tool like this is likely to distract me from my development.
- **F5** - I preferred the use of the tool On Commands versus On Save
- **F6** - I would trust the tool

For this scale, we will use the same scoring system, based on their answers from F1 to F6 in both the usability questionnaire and the general public questionnaire. On a scale of -2 (Strongly Disagree) to 2 (Strongly Agree), answers were added, and divided by the number of participants, due to the difference in the number of participants, resulting in the final score. Intrinsically, this will equate to a minimum possible score of -2, a maximum score of 2, and an average of 0.

Secondly, participants will be presented with a list of features that were either already developed, or defined as future work. From this list of features, participants who experienced the tool first hand had to provided a selection of the most important features, and, subsequently, a selection of features they consider that could be improved.

- Support for JavaScript/TypeScript
- Number of code mutations attempts.
- Performance.
- Number of possible fixes.
- Complexity of suggested fixes.

- Adaptability to other IDEs.
- Support for all function types (exported, arrow functions, methods, etc.).

Though, participants who partook in the general public questionnaire did not test or personally experience our solution, as such, they were presented with small *gifs*⁶, and, from their analysis of the videos, were asked to, assess their comfort level with a tool like the one presented, and to list which features they considered to be the most important. In the selection of the most important features, participants had to answer at least one feature for each of the questions.

5.5.4 Usability Questionnaire

We will firstly start by analysing the questionnaire that was carried out by 16 students, all finalists of the Integrated Master in Informatics and Computing Engineering.

In Section 5.3.2 (p. 48) we defined the procedure to compare developer performance while using the pAPRika Visual Studio Code extension and without using it. In Section 5.4.2 (p. 52) we further detailed the tasks of each of the problem sets. And, to evaluate developer performance 3 metrics were taken into consideration.

- **Time** to reach a **solution**.
- Final **code** of the solution.
- Whether or not the extension was used.

5.5.4.1 Participant's Characterisation and Background

As all participants of this study were finalists of the Integrated Master in Informatics and Computing Engineering, 100% of the participants answered “Bachelor's degree” when asked about the highest completed degree of education. Regarding years of professional experience, out of the 16 participants, 14 answered 0, while 2 answered 3.

While analysing the mean background scores from Table 5.1, we can observe that both groups appear to be above average regarding the languages, frameworks and technologies used in this experiment.

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

Group	Size	Mean	Std. Deviation	σ^2	<i>Shapiro-Wilk</i> (<i>p</i>)	<i>Levene</i> (<i>p</i>)	t-test (<i>p</i>)
A	8	3.25	3.20	10.21	0.85	0.90	0.5470
B	8	4.25	3.28	10.79	0.15		

As the obtained p-values for both the *Shapiro-Wilk* test and the *Levene* test are above the previously defined significance level of 0.05, we cannot reject neither of the null hypotheses. As such, we proceeded to calculate the *Student's t-test*, which, by conventional criteria, is considered

⁶pAPRika Website (Retrieved by: 30 June 2020)

to be not statistically significant, therefore, failing to reject the null hypothesis **H0**, accepting that no statistical difference can be observed.

5.5.4.2 Time to Solution

To evaluate the time to reach a solution of both groups, when participants use the tool or not, we set out to calculate the mean, standard deviation, and two-sided *Mann–Whitney U* test. As Campos has put it, the two-sided *Mann–Whitney U* test will allow us to understand the significance of the difference between both groups, by testing the following hypotheses [Cam19]:

H0 (Null Hypothesis): *The distributions of the populations from which each group was sampled are identical.*

H1 (Alternate Hypothesis): *The distributions of the populations from which each group was sampled are not identical.*

Table 5.2: Statistical measures and Mann–Whitney U (p) of time to solve each problem set.

Task	Set	Tool	Mean	Std. Deviation	Mann-Whitney U (p)
1	X	Yes	00:34	00:24	<u>0.00094</u>
		No	03:17	01:48	
	Y	Yes	00:48	00:41	<u>0.02382</u>
		No	02:59	02:31	
2	X	Yes	00:32	00:24	<u>0.00138</u>
		No	03:08	02:02	
	Y	Yes	00:33	00:17	<u>0.00094</u>
		No	02:50	01:10	
3	X	Yes	04:07	01:46	<u>0.02382</u>
		No	06:30	01:38	
	Y	Yes	04:14	01:28	0.06576
		No	06:12	01:47	
4	X	Yes	01:19	00:32	0.34212
		No	01:59	01:15	
	Y	Yes	00:35	00:15	<u>0.00736</u>
		No	01:25	00:34	

From a quick analysis to Table 5.2, we can observe that developers who use the tool are prone to be faster at finding and solving bugs. This difference is exacerbated in the **immediate** types of problem, *i.e.* tasks 1 and 2, and less meaningful in **nonimmediate** and **nonpresent** types of problems, *i.e.* tasks 3 and 4, respectively (*cf.* Section 5.4.2, p. 52).

Such remarks go in line with our expectations, as immediate tasks, when running the tool, require nothing more than opening the file, to immediately have a suggested fix, while other tasks required participants to continue, or start a new, implementation before providing the developer with a bug fix suggestion.

Furthermore, we can reject the null hypothesis **H0** that the distributions of the populations of tasks 1, 2, 3 (problem X), and 4 (problem Y) were sampled identically, while for the remaining two

tasks we cannot reject **H0**, and are incapable of establishing statistical differences between the two groups.

5.5.4.3 Code and Extension Use

By virtue of the problems themselves, since there is only a need for slight modifications to successfully tackle them. As such, ordinary code quality metrics do not serve a significant purpose, nor were easy to measure for remote tests, and, thus, participants were only required to pass the tests, with a bond of trust between the overseer and the participant. Participants in the end also reported their own times, shared some of the difficulties while solving the problems, and the overseer asked a number of questions to ensure the validity of the test. All participants reported having used the tool in all problems, except for one participant, who did not use the tool in task 3, in the problem set which its use was allowed.

During this experiment, there was one participant who failed to correctly understand that the extension had to be disabled where the part of the test required to do so, therefore, his participation had to be discarded.

5.5.4.4 Participants' Task Evaluation

In both group A and B, after solving the tasks, participants were required to fill a brief group of Likert-type items concerning their experience both with and without the tool. Items from P1 to P4 were asked in parts that the use of the tool was not allowed, and items P1 to P8 were used in parts that the use of the tool was allowed.

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.

By applying the same scoring algorithm used in the participants' background, where, on a scale of -2 (Strongly Disagree) to 2 (Strongly Agree), however, instead of totalizing the scores per participant, adding together the scores of all participants who apply (eight per group), i.e. obtaining a comparable final score. Intrinsically, this equates to a minimum possible score of $8 \text{ participants} * -2 = -16$, a maximum score of $8 \text{ participants} * 2 = 16$, and an average of 0. However, due to the straightforwardness of the problems presented, we expect all scores to be positive by a good margin.

From Table 5.3 (p. 61), we can extract that, overall, the use of the tool greatly aided participants in finding bugs, but also to find solutions in a faster and more streamlined way.

Table 5.3: Scoring of questions per problem, according to the use of the tool.

Set	Tool/Group	Question	Score
X	Yes/A	P1	9
		P2	6
		P3	15
		P4	4
		P5	13
		P6	13
		P7	14
		P8	7
	No/B	P1	2
		P2	4
		P3	7
		P4	16
Y	Yes/B	P1	12
		P2	11
		P3	13
		P4	4
		P5	16
		P6	15
		P7	15
		P8	8
	No/A	P1	1
		P2	4
		P3	11
		P4	8

5.5.4.5 Participants' Final Remarks

Problem Set X

In this problem set, we can observe that when not using the tool, participants felt that the time spent identifying the bug, instead of fixing it, was too great.

We can also observe that participants that used the tool, although its suggestions were correct, did not blindly trust the tool and took their time analysing its output, as per one of our objectives with the tool.

Overall, participants when using the tool felt that their solutions are more correct when compared to when they did not use the tool.

Moreover, participants declare that their confidence in the produced solutions, without the tool, is largely inferior when compared to participants that used the tool.

Feedback from the tool (P5-P7) was greatly positive.

Problem Set Y

In Problem Set Y, although users without the tool felt that bugs were not that easy to identify and that solutions were not straightforward, their confidence in the produced solutions was higher compared to problem set X. However, while using the tool, participants felt great

ease in finding bugs, and a better understanding of the solutions, which also had a positive impact in their confidence of the produced solutions.

Feedback from the tool (P5-P7), similarly to problem set X, was greatly positive.

Post-Test Feedback

To assess the participants' comfort level with the tool and its features, we proceeded to calculate the comfortability score with questions F1 to F6 (*cf.* Section 5.5.3, p. 57).

Given that the comfortability score scale of Table 5.4 goes from -2 (Strongly Disagree) to 2 (Strongly Agree) we can infer the following:

- **F1** - Participants agree that the tool's features were simple to use and easy to understand.
- **F2** - Participants agree that the tool can positively impact their development workflows.
- **F3** - Participants would consider using the tool.
- **F4** - Participants disagree that a tool like this is likely to distract from their development.
- **F5** - Participants have no preference on how to run the tool.
- **F6** - Participants somewhat agree that they would trust the tool.

Table 5.4: Comfortability score, per participant, for questions F1 to F6, based on the use of this tool.

	F1	F2	F3	F4	F5	F6
Average Score	1.56	1.38	1.44	-0.94	-0.25	0.88
σ	0.63	0.50	0.51	0.57	1.06	0.81
σ^2	0.40	0.25	0.26	0.33	1.13	0.65

As also declared in Section 5.5.3 (p. 57), participants who experienced the tool first hand provided a selection of the most important features, and, subsequently, a selection of features they consider that could be improved.

From Figure 5.7 (p. 63) we can infer that, overall, developers that partook in this study enjoyed the tool's support for JavaScript/TypeScript, number of code mutations attempts and possible fixes, performance, complexity of suggested fixes, and function types support. As the adaptability for other IDEs is still under development, and participants could only experience the tool in Visual Studio Code, they did not consider it as a current important feature.

In contrast, in Figure 5.8 (p. 63), participants perceived the complexity of suggested fixes, the number of possible fixes and, as expected, the adaptability to other IDEs, as the features that required further work.

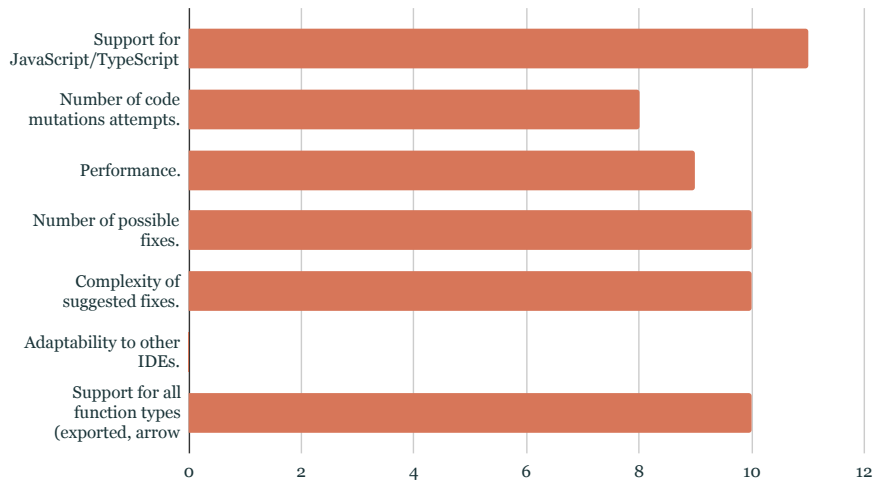


Figure 5.7: Bar chart with the participants’ most important features of the tool.

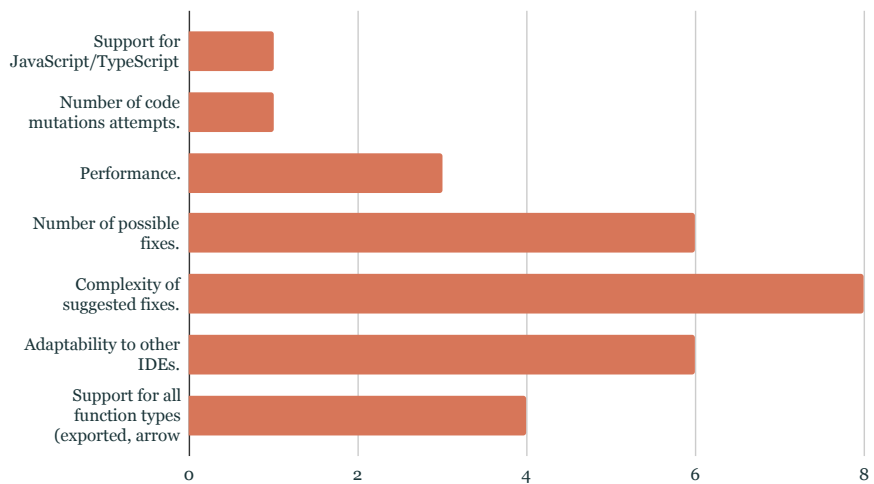


Figure 5.8: Bar chart with the participants’ features of the tool that should be improved on.

5.5.5 General Public Questionnaire

Participants for this questionnaire will all be from various curricular years of the Integrated Master in Informatics and Computing Engineering. We will start by taking a brief look at these participants’ characterisation and background, and, then, proceed to evaluate each of the tasks as described in Section 5.4.1 (p. 50).

5.5.5.1 Participant’s Characterisation and Background

As we can observe from Figure 5.9 (p. 64), most participants highest completed degree of education is “Bachelor’s degree”. This may be due to a number of factors, *e.g.* greater confidence of more experienced students to answer to questionnaires.

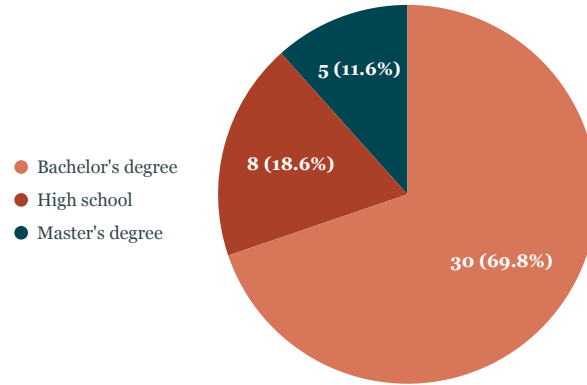


Figure 5.9: Pie chart with the highest completed degree of education of participants.

Table 5.5: Statistical measures of background scores.

Size	Mean	Std. Deviation
43	4.16	4.50

Analysing the mean background scores from Table 5.5, we can observe that both groups appear to be above average regarding the languages, frameworks and technologies used in this experiment.



Figure 5.10: Histogram with the number of participants per years of professional experience.

Regarding the years of professional experience, we can observe that most participants have zero years of professional experience, a common characteristic in university students in Portugal [REN19].

5.5.5.2 Task 1

In this first task, we presented participants with two sets of example-based tests, one for `myParseInt` (cf. Figure 5.1, p. 51) and `longestCommonSubstring` (cf. Figure 5.2, p. 51).

Our objective for this task was to evaluate the participants' ability to think of more example-based tests in addition to the ones already defined.

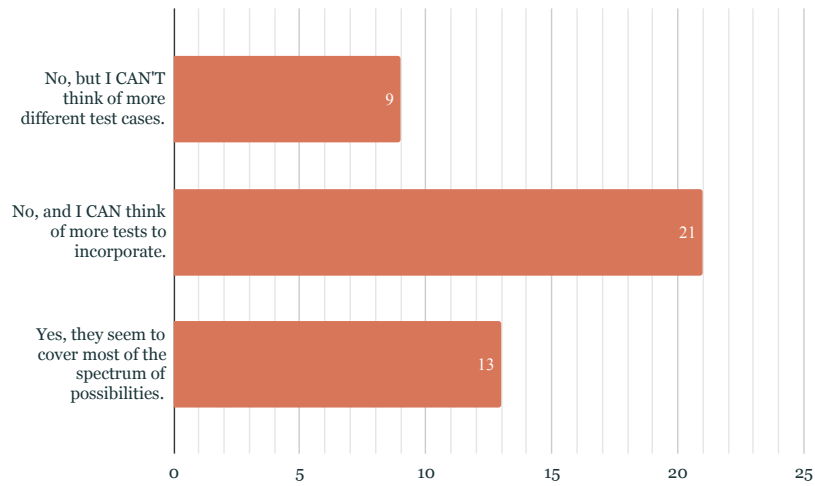


Figure 5.11: Bar chart for task 1's `myParseInt` assessment with participants per selected answer.

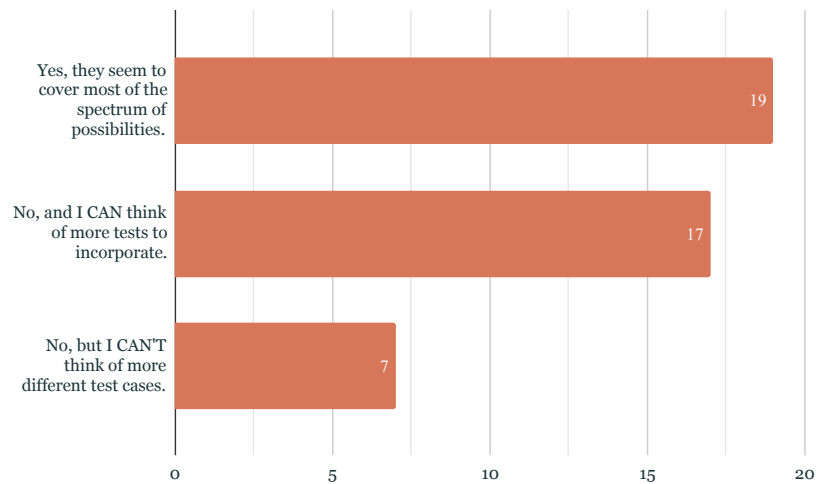


Figure 5.12: Bar chart for task 1's `longestCommonSubstring` assessment with participants per selected answer.

From the results obtained in both Figure 5.11 and Figure 5.12, we can then proceed to assume that the participants' ability to think of example-based tests, namely edge cases, should improve greatly, in order to consider example-based tests as specifications for test-driven development. In short, for our population, example-based tests may not as reliable as once thought.

5.5.5.3 Task 2

In the second task, we present participants with a brief description of what a property is, and with properties of both `myParseInt` (cf. Figure 5.3, p. 52) and `longestCommonSubstring` (cf. Figure 5.4, p. 52).

The given examples were both incomplete, however, the longest common substring algorithm is more complex, and, thus, more complicated for participants to think properties of. With the given examples, participants were then asked, for both examples, whether or not they considered that they could have come up with the given properties.

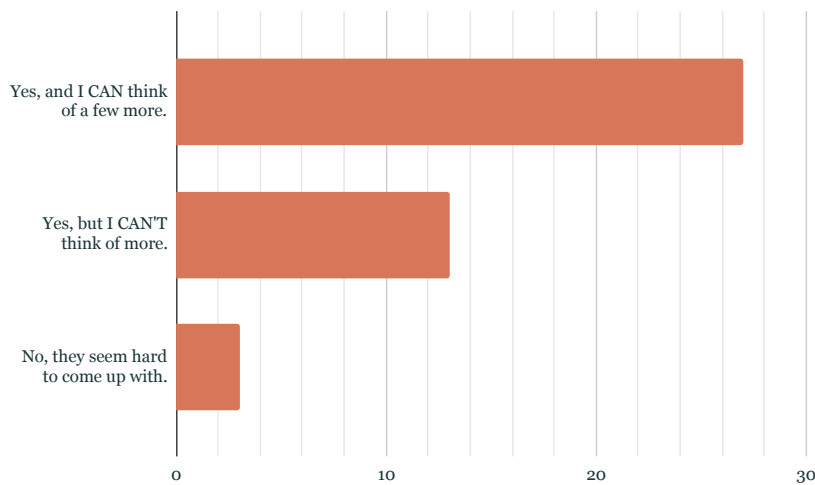


Figure 5.13: Bar chart for task 2's `myParseInt` assessment with participants per selected answer.

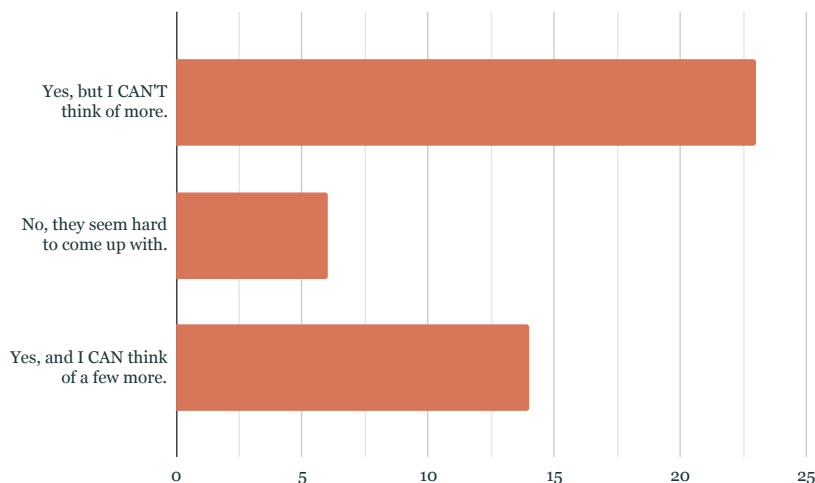


Figure 5.14: Bar chart for task 2's `longestCommonSubstring` assessment with participants per selected answer.

From the results obtained in both Figure 5.13 and Figure 5.14, we can then confirm our

conjecture that participants would face an increased difficulty in producing additional properties for `longestCommonSubstring`, unlike `myParseInt`, where most participants could generate more properties for the given function.

5.5.5.4 Task 3

Finally, in the first task, we present participants with an implementation of the properties given in task 2 in a property-based testing framework - in this case, *fast-check* - of both `myParseInt` (cf. Figure 5.5, p. 53) and `longestCommonSubstring` (cf. Figure 5.6, p. 53).

Based on the given examples, participants were asked, assuming that the example-based tests were incomplete and that property-based testing would widen the spectrum of test possibilities, to consider if, for both examples, they would write property-based tests.

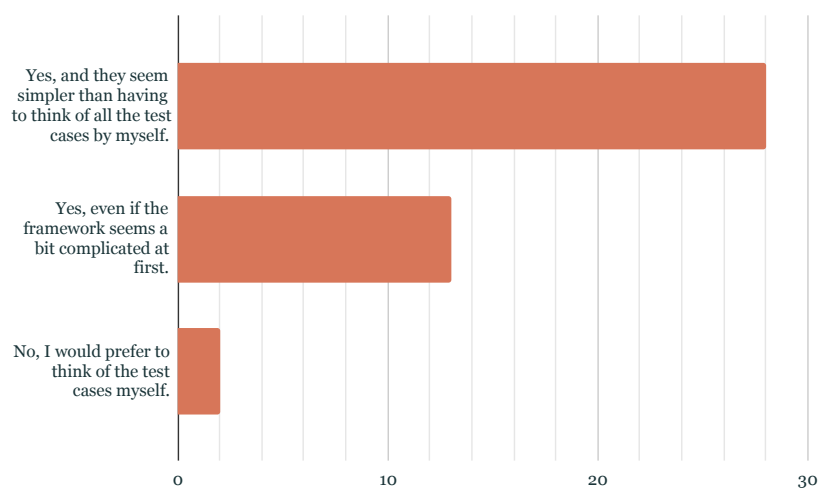


Figure 5.15: Bar chart for task 3's `myParseInt` assessment with participants per selected answer.

From the results obtained in both Figure 5.15 and Figure 5.16 (p. 68), we can observe a great enthusiasm towards learning property-based testing, even if some consider that the example framework seemed complicated at first.

5.5.5.5 Participants' Final Remarks

Post-Test Feedback

To assess the participants' comfort level with the tool and its features, as presented in the gif, we proceeded to calculate the comfortability score. An analysis of Table 5.6 (p. 68), will be detailed in Section 5.7 (p. 71). Though, these participants, who partook in the general public questionnaire, did not test or personally experience our solution, as such, their evaluation was based on the aforementioned *gifs* (cf. Section 5.5.3, p. 57).

Given that the comfortability score scale of Table 5.4 (p. 62) goes from -2 (Strongly Disagree) to 2 (Strongly Agree) we can infer the following:

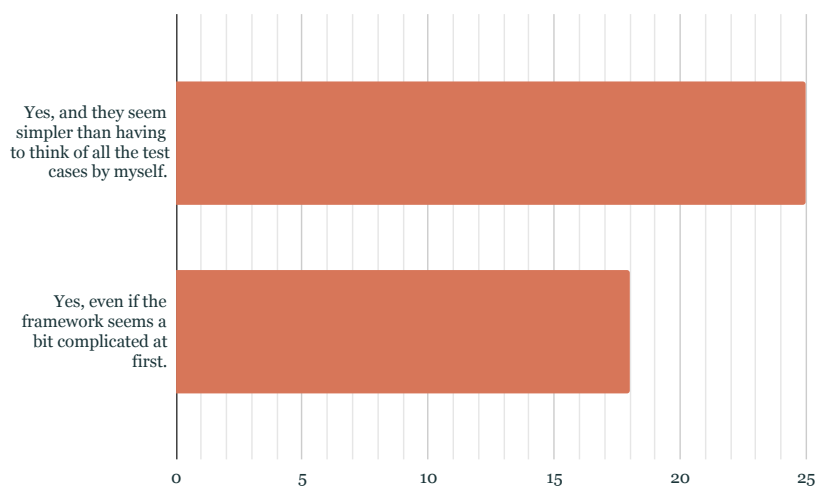


Figure 5.16: Bar chart for task 3's `longestCommonSubstring` assessment with participants per selected answer.

- **F1** - Participants agree that the tool's features were simple to use and easy to understand.
- **F2** - Participants somewhat agree that the tool can positively impact their development workflows.
- **F3** - Participants would consider using the tool.
- **F4** - Participants somewhat disagree that a tool like this is likely to distract from their development.
- **F5** - Participants have no preference on how to run the tool.
- **F6** - Participants somewhat agree that they would trust the tool.

Table 5.6: Comfortability score, per participant, for questions F1 to F6, based on the provided gifs.

	F1	F2	F3	F4	F5	F6
Average Score	1.07	0.88	1	-0.81	-0.07	0.59
σ	1.07	0.89	0.96	1.08	0.87	0.97
σ^2	1.15	0.79	0.92	1.16	0.76	0.94

From their analysis of the *gifs*, these participants also provided a selection of what they considered to be the most important features of such a tool.

From Figure 5.17 (p. 69) we can infer that, overall, developers that partook in this study seemed excited about our tool, namely its support for JavaScript/TypeScript, number of possible fixes, performance, complexity of the suggested fixes.

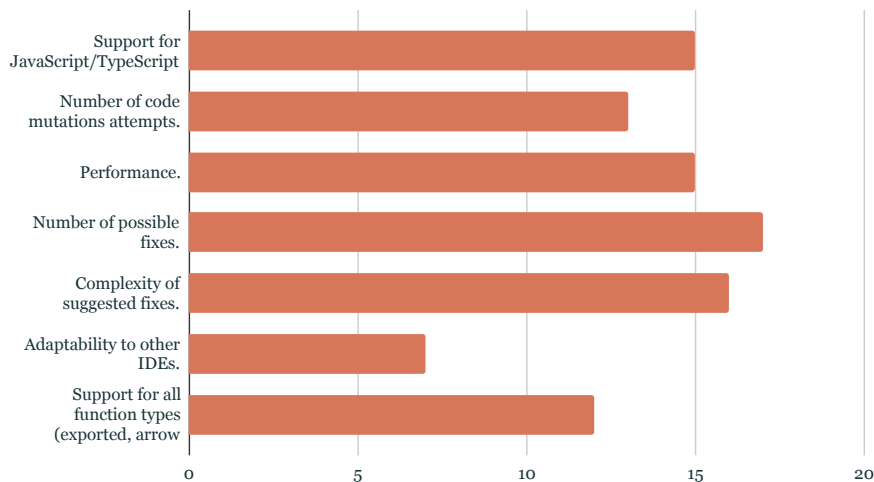


Figure 5.17: Bar chart with the participants’ most important features of the tool.

5.6 Threats to Validity

As an empirical study, a number of threats to validity were considered, as such we set out to, in this section, describe the threats to Construct Validity, Internal Validity and External Validity, and briefly discuss them.

5.6.1 Construct Validity

The first group of validity threats we present is the construct validity, which pertains to how the selection of formulations of the dependent and independent variables, i.e. tasks, environment, data collected and other variables, which may impact the quality and validity of our findings [AC05].

Confounding Constructs and Levels of Constructs

The validity of the tasks executed in our questionnaires is one of the most important and pressing threats. These must accurately represent, to a certain degree, buggy programs that a developer is likely to face daily. Furthermore, many developers prefer to own the proposed fixes, rather than simply applying them, which may raise questions about the sociology of APR [MBC⁺19]. As such, we set out to define three different types of problems (*cf.* Section 5.4.2, p. 52), which required different types of approaches, and a tool that did not simply apply the suggested fixes, but simply present them to the developer.

Hypothesis Guessing

Participants, knowingly participating in a study, might consciously or subconsciously try to guess the hypothesis and attempt to perform accordingly [WRH⁺12]. Not everyone participates in research project passively, some try to reverse-engineer the study, and adjust their behaviours accordingly. Even though it is not possible to assure that a participant is not acting honestly, we opted to not reveal the hypothesis and objectives of the study, and

assured that participants consented that their participation was honest and critical, in an effort to minimize the effect of this threat.

Evaluation Apprehension

As some people act anxiously when being evaluated, their behaviour might lead to poor performance, or, lead to an attempt of wanting to appear “smart”, we made sure that participants were taking part in this study willingly, avoiding any kind of coercion, social pressure, or mere pressure during the tasks.

5.6.2 Internal Validity

The second validity group of validity threats, is none other than the Internal Validity, which analyses whether or not cause-effect relationships may be determined, among independent variables and observed effects in dependant variables [WRH⁺12].

Development Environment

Nitpicking developers might have a certain development environment configured that aims to augment their productivity. However, even the least nitpicky developer is accustomed to their development environment, and, as such, we decided not to introduce a remote testing environment, as health-related circumstances would impose, forcing participants to use a certain configuration, as we believe would have no impact on the results, even if developer performance increased. Such can be verified by the similarity of values between Campos work, and our own.

Physical Environment

Once again, due to certain limitations imposed by a worldwide pandemic, we could not create a closed and controlled physical environment where participants could partake in the test. However, as participants were all at the comfort of their own home, and could participate in the experiment with a very open schedule, noise or other distracting factors were always excluded by the participants.

Task's difficulty

The benchmark created by Campos [Cam19] was created to test the usability of the tool, instead of setting itself as a difficult set of tasks for the participants. However, even if the created tasks were of low complexity and objectively required little changes for a complete fix of the bug, some participants had difficulty solving some tasks, which we believe do not affect the results obtained with the said benchmark.

5.6.3 External Validity

Finally, we will introduce the External Validity section, concerning the degree to which the conclusions of a study can be extended to different populations or settings [WRH⁺12].

Sample Size

In our empirical experiments, the general public questionnaire and the usability questionnaire, we reached 27 participants and 16 participants, respectively. Certain sections of the

usability questionnaire could also be exported to the general public questionnaire, totalling 43 participants in the new tasks defined by our study (*cf.* Section 5.4.1, p. 50). The sample size was always taken into account when interpreting the findings of this study.

Sample Characteristics

Because of the restrictions imposed by the nature of the usability questionnaire itself, finding participants who are willing to forfeit some of their precious time was a difficult task, which led to a population comprised of solely students of the final year of the Integrated Masters of Informatics and Computing Engineering at the Faculty of Engineering of the University of Porto. The population of the general public questionnaire, on the other hand, is much more diverse, while still having software engineering knowledge. Regardless, as certain findings state, students do not perform better nor worse than software engineering professionals when using new technologies during experimentation [SMJ15].

5.7 Discussion

The development, implementation and experimentation were all performed to validate the following hypothesis:

“Using Property Tests as Specifications in an Automated Program Repair tool helps to eliminate overfitting.”

Such a statement can be subdivided into different specific elements, serving as the building block of our hypothesis, *i.e.* the research questions:

- **Research Question 1** - *“Are developers capable of understanding and formulating property-based tests?”*

In order to understand the viability of using property-based tests as specifications, we set out to evaluate whether or not participants can easily understand and formulate property-based tests.

With this in mind, we created Task 2 and Task 3 (*cf.* Section 5.4.1, p. 50), which evaluated the comprehension of a set of given properties, and their ease of implementation within a PBT framework.

The results obtained in Task 2 (*cf.* Section 5.5.5.3, p. 66) show an overall understanding of properties. However, in more complicated examples, such as the `longestCommonSubstring`, users have a harder time thinking of more properties, which reinforces the barrier that PBT defines, as it establishes itself as the robust byproduct, and reward, of a more considered upon testing suite. While results obtained in Task 3 (*cf.* Section 5.5.5.4, p. 67) demonstrate a high approval of the use of PBT frameworks with a portion of participants declaring they would need more training with the framework.

- **Research Question 2** - “Do developers believe in the completeness of example-based tests?”

To evaluate participants’ belief in example-based test suites, we presented participants with two *incomplete* test suites, for JavaScript’s `parseInt` and the known algorithm `longestCommonSubstring` (cf. Section 5.5.5.2, p. 64).

Subsequently, we asked participants whether they considered these test suites, knowingly incomplete, complete or not, and, although we were expecting a portion of participants to find the test suites complete, we were not expecting such percentages of 30% (cf. Figure 5.11, p. 65) and 44% (cf. Figure 5.12, p. 65), for `parseInt` and `longestCommonSubstring`, respectively. Furthermore, 21% and 16%, respectively, did find the tests incomplete, while, however, admitting not being able to think of more test cases.

In short, the percentage of our participants capable of fully thinking of more test cases for our examples is too small, proving the need for property-based testing.

- **Research Question 3** - “Can property-based testing truly solve overfitting?”

Due to the own nature of property-based testing, where properties are defined and a generative-testing engine creates randomized inputs to verify such properties, and since our tool’s suggestions are based on completely passing the test suites, and property-based test suites *do* find more bugs⁷⁸; we believe that the use of property-based testing as specifications towards automatic program repair is capable of augmenting the number of bugs found, and, in turn, augment the reliability of patches, as even our incomplete properties were able to be more complete than the example-based test suites (where a big portion of users considered complete). Therefore, we believe *overfitting* truly can be solved by PBT.

Table 5.7: Levene test and Student t-test, for questions F1 to F6, between both questionnaires.

	F1	F2	F3	F4	F5	F6
Levene (p)	0.29	0.39	0.59	0.039	0.11	0.26
Student t-test (p)	0.050	0.026	0.050	0.33	0.27	0.16

- **Research Question 4** - “Do developers believe a live Automatic Program Repair technique is easy to use?”

To evaluate ease of use of the tool and its features, we extracted the comfortability score from question F1 (cf. Section 5.5.3, p. 57), for both the usability questionnaire and the general public questionnaire.

In both Table 5.4 (p. 62) and Table 5.6 (p. 68), we can observe that developers, overall, feel that the **tool is easy to use**. Developers that truly experienced the tool, however, demonstrated a more consistent understanding of its features and ease of use, when compared to developers who only had access to the *gifs* demonstrating the use of the tool.

⁷Issues discovered using fast-check (Retrieved by: 2 July 2020)

⁸Spotify: Generating test cases so you don’t have to (Retrieved by: 2 July 2020)

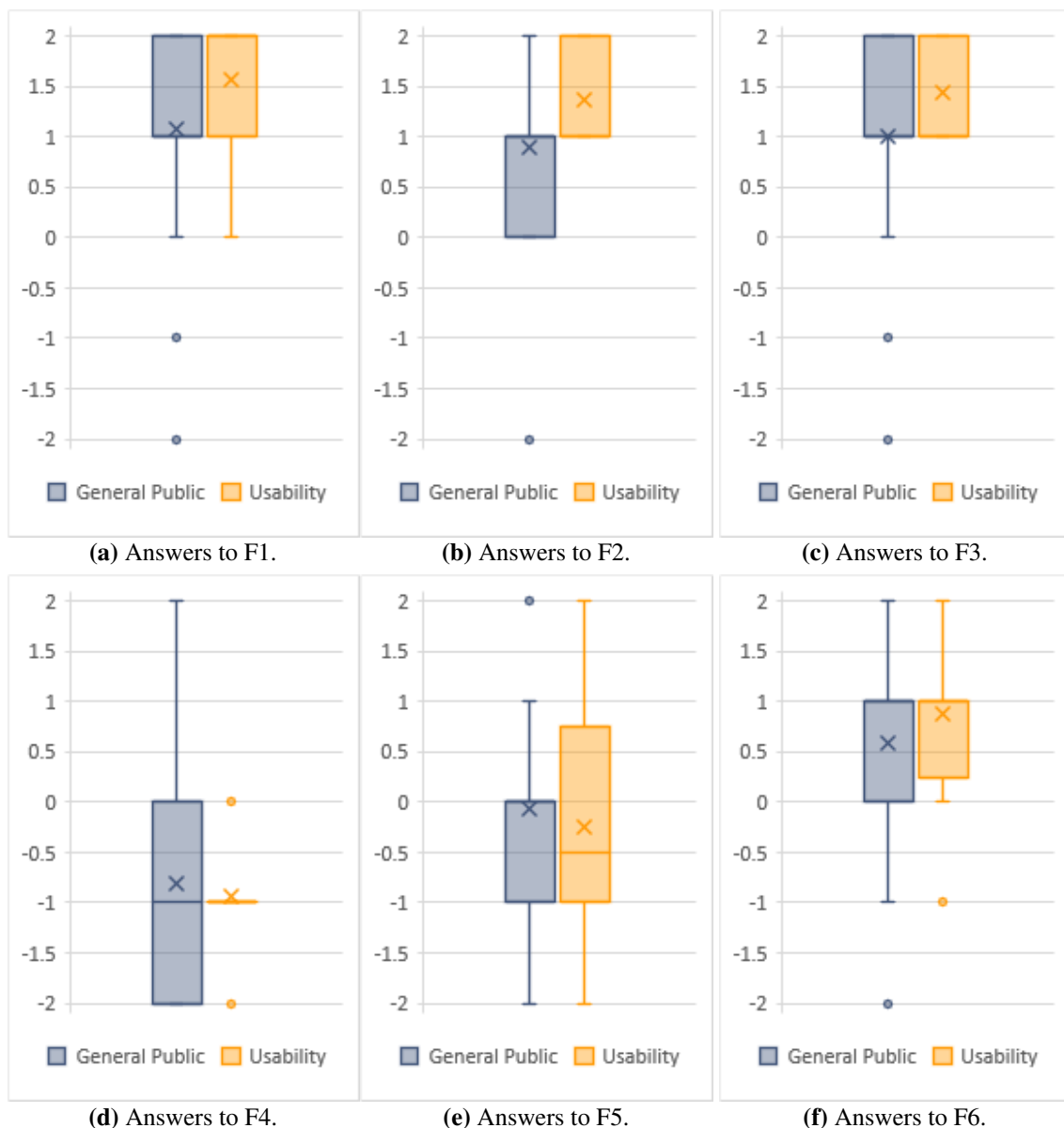


Figure 5.18: Answers to the post-test section of both questionnaires. Overall we can see stronger agreement on the results of the usability questionnaire through questions F1-F3 and F6. For question F4 we can see an overall agreement that the tool is not as distracting as it seemed to the General Public. Finally, for question F5 we can see a higher disagreement after using the tool due to it being a matter of preference, which is reinforced through its use.

When observing the statistical difference tests performed for question F1 in Table 5.7 (p. 72), we can conclude that the homogeneity requirement is met through the *Levene's test*, and, when looking at *Student's t-test*, we can observe that the result is significant (considering that $p = < 0.05$ is significant). As such, we can assume that there is a slight scepticism towards the ease of use of the tool without experiencing it, which after experiencing it, mostly disappears. Such differences can also be observed through Figure 5.18.

- **Research Question 5** - “Can developers see a positive impact on their workflow through the use of a live Automatic Program Repair technique?”

To evaluate whether or not developers could see an improvement to their workflow from the use of this tool, we extracted the comfortability score from questions F2 and F3 (cf. Section 5.5.3, p. 57), for both the usability questionnaire and the general public questionnaire.

In both Table 5.4 (p. 62) and Table 5.6 (p. 68), we can observe that developers, overall, can see a **positive impact** caused by our tool in their development workflow, and a similar interest in using the tool. Developers that truly experienced the tool, however, demonstrated a more consistent view of these aspects, when compared to developers who only had access to the *gifs* demonstrating the use of the tool.

When observing the statistical difference tests performed for questions F2 and F3 in Table 5.7 (p. 72), we can conclude that the homogeneity requirement is met through the *Levene’s test*, and, when looking at *Student’s t-test*, we can observe that the result is significant for both F2 and F3 (considering that $p = < 0.05$ is significant). As such, we can assume that there is a slight scepticism from users who did not use the tool, however, after experiencing the tool, developers have an overall stronger agreement on its positive impact, which can also be observed through Figure 5.18 (p. 73).

- **Research Question 6** - “Do developers trust the capabilities of a live Automatic Program Repair technique?”

To evaluate the developers’ trust in the tool and its features, we extracted the comfortability score from question F6 (cf. Section 5.5.3, p. 57), for both the usability questionnaire and the general public questionnaire.

In both Table 5.4 (p. 62) and Table 5.6 (p. 68), we can observe that developers, overall, trust the capabilities of the tool. Developers that truly experienced the tool, however, demonstrated a **stronger trust** in the tool, when compared to developers who only had access to the *gifs* demonstrating the use of the tool.

When observing the statistical difference tests performed for question F6 in Table 5.7 (p. 72), even though the homogeneity requirement is met through the *Levene’s test*, when looking at *Student’s t-test*, we can observe that the result is not significant. However, there is still a difference that can be observed between Table 5.4 (p. 62) and Table 5.6 (p. 68), which can be better observed through Figure 5.18 (p. 73).

Furthermore, as validation of both Campos’ work [Cam19] and our own, we have decided to take a deeper look at the research questions of the author, while putting our obtained results to the test:

- **Research Question 7** - “Are users faster in reaching the solution when using a live Automatic Program Repair tool?”

Similarly to Campos’s results, in Section 5.5.4.2 (p. 59), we end up rejecting the null hypothesis for most, i.e. there are no statistical differences between using and not using the tool.

However, relevant statistical differences can still be observed in Table 5.2 (p. 59) in most tasks, and the mean time to reach a final solution is consistently lower when using the extension. As such, we corroborate Campos’ conviction that there exists substantial evidence that users are faster to reach a solution when using a live Automatic Program Repair tool, a clear improvement when comparing to the results obtained by other tools [CSC⁺19].

- **Research Question 8** - “*Are solutions generated by an Automatic Program Repair tool different from the ones developed by human programmers?*”

In our study, due to the limitations imposed by a global pandemic, proper verification and analysis of the code produced by each of the participants was impracticable. As such, we were not able to neither properly compare solutions between the ones purely generated by humans *versus* the ones generated by the tool, nor appoint the best and worse code.

However, since our solution produces various suggestions per problem, some are considered *unnatural* and were largely ignored by the developer.

We believe that an improved benchmark might be able to better measure such metric, while also reckoning that improvements to the tool may also be implemented to opt for more *natural* suggestions and discarding *unnatural* ones.

- **Research Question 9** - “*Are users aware of the rationale of solutions generated by the Automatic Program Repair tool before accepting them?*”

As in Campos’ work, we were not able to establish a clear understanding of the participants’ tendency to understand the solutions before accepting them.

In fact, by analysing participants’ answers to question P8 (*cf.* Section 5.5.4.4, p. 60) we are able to infer that participants agreed that achieving a level of comprehension of the suggestion was attempted. However, we believe results for this question may be slightly biased, due to the presence of **immediate** problems in tasks 1 and 2, since it allowed participants to obtain a solution in a time that we believe is too short to read through the problem and the faulty code, and understand the suggested repair. Evidence of such bias may be the extremely low average times to find a solution, which can be corroborated by Campos’ results.

As with **RQ5**, we believe an improved benchmark, with more detailed per task questions, might obtain better results in this aspect.

5.8 Summary

Throughout the Empirical Evaluation chapter, we have presented the objectives and guidelines for our experiments (*cf.* Section 5.1, p. 45), immediately followed by the planning for each one

of the experiments (*cf.* Section 5.3, p. 47). In Section 5.4 (p. 50) we closely describe the tasks at hand for both questionnaires developed to validate our hypothesis. It is then immediately followed by Section 5.5 (p. 55), where we present the obtained results, explain some of the methods used, accompanied by an analysis and interpretation of said results. We then analyse the threats to validity (*cf.* Section 5.6, p. 69) of the study, while referring to some of the steps taken against them. Finally, in Section 5.7 (p. 71), we present a critical discussion of the results, in regards to the hypothesis and research questions posed.

Chapter 6

Conclusions

6.1	Summary	77
6.2	Main Contributions	78
6.3	Future Work	79

6.1 Summary

As software is increasing in size, complexity, and intractability, faster and more efficient development processes must be defined to account for the higher demand in quality, performance, robustness, and maintainability. We have also established that even though the scientific community has developed a myriad of studies and tools to address this demand in the form of automated program repair solutions. However, as presented in Section 3.2 (p. 20), several issues can still be found, even in the solutions developed in the past years. Even though, every attempt at automatic program repair has been moving the state of the art forward, it is considered that APR is another AI-complete problem [OS19], requiring high computational difficulty that only a strong AI would be able to truly solve. Furthermore, most APR tools have been predominantly been restricted to the academic environment, limiting its dissemination and use by the general public, and the number of tools available online is even smaller, with an almost non-existent presence in the open-source scene.

As such, the purpose for this dissertation was threefold, that is, to gather the state of the art on the topics of automatic programming, unit testing, language server protocols, and respective tools; to create a selection of suitable approaches to automatically generate code completion suggestions based on existing specifications, in the form of properties; and, finally, to develop a plugin capable of suggesting smart semantic auto-completion, using live testing, to the developer, enabling both automated rectification of bugs and vulnerabilities, and accelerated development.

Furthermore, we set out to produce fixes that are acceptable to the programmer and avoid overfitting to test cases by the use of PBT as its specifications. Since, to achieve specification

completeness, in example-based testing, the tests' creator has to think of all the possible cases, though we argue that if such test cases can be thought of, they are probably already well regarded during their development, ending up with success during the testing phase. Taking this into account, our solution does not attempt to apply fixes without a developer, as it is a mere tool that intends to help the developer's reasoning, i.e. we do not attempt to create a true panacea. Additionally, our tool supports PBT, which we believe eliminates the overfitting problem, and does not attempt to be too greedy with its mutations, instead, it opts to find and repair the small bugs that developers, a lot of the times, fail to discover.

In order to validate our hypothesis (cf. Section 3.3, p. 22), we decided to perform an empirical study (cf. Chapter 5, p. 45) in which had two main purposes, to obtain results that allow a more direct comparison between pAPRika users *versus* users that solely use normal test-driven development with example-based test suites; and to obtain reactions from the participants to property-based testing as a component in their development workflow.

“Using Property Tests as Specifications in an Automated Program Repair tool helps to eliminate overfitting.”

Finally, to validate the hypothesis above, we attempted to answer the enumerated research questions in Section 5.7 (p. 71), which arrived at the following conclusions. Participants are somewhat incapable of writing complete example-based tests, however, they demonstrate an understanding and capability of formulating property-based tests. Furthermore, there is wide acceptance of PBT frameworks, with a small portion of participants declaring a need for more training with the framework. And, finally, even though the overall acceptance of the tool was positive, we were able to observe lower trust and higher scepticism from participants who did not experience the tool, whilst the other participants have a higher trust in its functionalities, due to their experience with it. Such results have given us confidence over the product developed, as previous studies with other APR tools have obtained results in which developers did not trust the tool [RAW⁺19, AWG⁺20].

6.2 Main Contributions

In Section 3.2 (p. 20) we listed some of the problems that affect current APR tools. In this work, we do not attempt to fix them all, however, we believe that we still have made a valuable contribution to the automatic program repair scientific community. This contribution can be summarised in the following points:

- A ready-to-be open-sourced IDE Extension employing a live Automated Program Repair tool, which leverages the power unit tests (example-based or property-based), while using a mutation generation APR technique to generate fix suggestions to codebases where faults were found. In addition, our tool is capable of presenting multiple suggestions, which we believe enables the suggestion of more natural fixes, and, paired with the greatly increased

feature set described in Section 4.3 (p. 27), turns our solution into one that truly finds its purpose and meets its need in the test-driven development process.

Although we have increased the suggestion power of our tool, which is able to better help with the developers' reasoning, it does not sacrifice its real-time intervention, maintaining a level of 5 in the liveness scale[Tan13], providing tactically predictive feedback.

- Expanded the results obtained with Campos' empirical Study with an additional 16 participants, and expanded its scope to include an evaluation of participants' reactions to property-based testing as a component in their development workflow. In this evaluation participants were tested for their ability to think of example-based tests, to come up with properties, and to integrate them in PBT frameworks.
- Small contribution, as a 4th author, to a paper that was submitted to the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020).

6.3 Future Work

As a result of this research project, pAPRika, the tool developed in this dissertation, has many improvements that can be explored and turned into future work. One of the final goals of this tool is to become open-source, and, as such, many steps during this work were taken to ensure that this project could become a good open-source project, transforming pAPRika into a popular tool for any IDE. Thus, to better ensure that this tool achieves its purpose, some features were thought of as to improve our solution:

- **Support for other IDEs** - Development of more clients for our tool, in order to support more IDEs and expand the tool's presence in the developer community.
- **Bug description** - Implementation of *potential* descriptions for bugs found, so that developers have more insightful feedback of fix suggestions. Such can be achieved by either the assumption of what the fix is, or by a deeper analysis of the problem at hand.
- **Fix ranking** - Our tool already successfully finds buggy functions, generates patches, and validates such patches suggesting them as fixes. However, the fixes presented to the developer are an assortment of fixes that may or may not be considered *correct* to the human-eyes. As such, the development of a heuristic to rank the suggested fixes, to, then, only present the most *correct* is a feature that would greatly improve the tool's usability. Such a feature would likely have to be based in a heuristic, as we can find in similar approaches [BYP19, CPF17, KLB⁺19], effectively presenting developers with the few top-ranked fixes, to more efficiently assess their correctness and applicability to the codebase. The experimental evaluation in Sec. IV comments on the effectiveness of JAID's ranking heuristics
- **Tests' status visual cues** - In Campos' implementation, certain visual cues were passed to developers, informing which tests passed and which didn't for the current codebase. This

functionality was initially discarded due to the adaptation of the tool within the LSP standard, however, it can be quickly re-implemented using a simple message between the client and server, as the responsibility of said action would lie on the client, if the support is present.

- **Tool testing** - This tool was developed with tests in mind, so it is only logical that our tool is also well tested to ensure its functionality. As such, some tests were developed, though, the coverage of these tests is still rather small, which motivates the creation of more tests, and possibly using PBT.
- **Further explore both PBT and APR state of the art** - For this dissertation various papers were analysed, however, due to the increasing number of APR solutions developed for the past years ¹, the result of these could not be compiled in time for a more complete state of the art chapter.

¹Program Repair Bibliography (Retrieved by: 2 July 2020)

Appendix A

General Public Questionnaire

pAPRika - Automatic Program Repair Extension

This survey aims to evaluate the usability of pAPRika, an IDE extension for JavaScript and TypeScript capable of finding bugs in your code and suggesting small fixes. This extension was developed in the context of a master thesis of the Master in Informatics and Computing Engineering at FEUP called "Property Tests as Specifications Towards Better Code Completion".

All answers are anonymous and will be used exclusively for academic purposes. If you're interested and consent, your participation will be acknowledged in our work.

Thank you for your collaboration,
Afonso Ramos

(Estimated completion time: 4-5 mins)

***Required**

Are you participating in this study willingly and do you consent that all opinions will be honest and critical? *

☐ Yes, and I give my consent.

Participant's Profile

This section will only be used to better understand the distributions among the participants. No filling is required if you don't want to share your personal information.

Education *

Highest completed degree of education. If currently enrolled in one, choose the highest one completed.

- ☐ High school
- ☐ Bachelor's degree
- ☐ Master's degree
- ☐ Doctorate's degree

Years of Professional Experience *

Your answer

Technical Background

Please describe your knowledge of software development, JavaScript, debugging, the use of testing frameworks, among others.

Answer according to your background. *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I have considerable experience with JavaScript.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with the Mocha testing framework.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with Visual Studio Code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with Test Driven Development.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I regularly use tools to help me code (linters, code completion, etc.).	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I am always capable of understanding code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I feel comfortable in identifying bugs in code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I feel comfortable in fixing bugs in code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Example-Based Test Cases

We will now present you with a set of test cases that enable the use of our tool.

With `myParseInt` one intends to implement a function capable of replicating JavaScript's own `parseInt`.

With `longestCommonSubstring` one intends to implement a function capable of returning the longest common substring between two strings.

Example-Based tests for `myParseInt`

```
describe('myParseInt', function () {
  it("should return same values as JavaScript's parseInt(). #fix {myParseInt} (1)", function () {
    expect(myParseInt('10')).toEqual(parseInt('10'))
  })
  it("should return same values as JavaScript's parseInt(). #fix {myParseInt} (2)", function () {
    expect(myParseInt('00 abs')).toEqual(parseInt('00 abs'))
  })
  it('should return NaN. #fix {myParseInt} (3)', function () {
    expect(myParseInt('abc 95')).toBe.NaN
  })
  it('should return NaN. #fix {myParseInt} (4)', function () {
    expect(myParseInt(' asd45.23')).toBe.NaN
  })
  it("should return same values as JavaScript's parseInt(). #fix {myParseInt} (5)", function () {
    expect(myParseInt('999.4')).toEqual(parseInt('999.4'))
  })
})
```

Do you think the tests above are enough for a developer to replicate JavaScript's `parseInt` behaviour? *

- ☐ Yes, they seem to cover most of the spectrum of possibilities.
- ☐ No, but I CAN'T think of more different test cases.
- ☐ No, and I CAN think of more tests to incorporate.

Example-Based tests for `longestCommonSubstring`

```
describe('Longest Common Substring', () => {
  it('should find "sentence" {longestCommonSubstring}', () => {
    assert.equal(
      longestCommonSubstring(
        'This is a sentence that will get analysed a lot',
        'There are a lot of sentences with a lot of common substrings'
      ),
      'sentence'
    )
  })
  it('should find "mportant" {longestCommonSubstring}', () => {
    assert.equal(longestCommonSubstring('Wording is important', 'Important wording'), 'mportant')
  })
  it('should find "ika" {longestCommonSubstring}', () => {
    assert.equal(longestCommonSubstring('pAPRika', 'paprika'), 'ika')
  })
})
```

Do you think the tests above are enough to always return the longest common substring? *

- ☐ Yes, they seem to cover most of the spectrum of possibilities.
- ☐ No, but I CAN'T think of more different test cases.
- ☐ No, and I CAN think of more tests to incorporate.

Property-Based Testing

Property-based tests are designed to test the aspects of a property that should always be true, rather than having to write a different test for every value that you want to test.

For the questions below please assume that there were missing test cases in the Example-Based tests above.

Defining Properties, a brief tutorial

Property-Based Testing relies on properties. It checks that a function, program or whatever system under test abides by a property. Most of the time, properties do not have to go into specific details about the output. They just have to check for useful characteristics that must be seen in the output.

A property is just something like:

for all (x, y, ...)
such as precondition(x, y, ...)
holds property(x, y, ...) is true

Property example:

for all (a, b, c) strings
the concatenation of a, b and c always contains b

However, these properties still need to be implemented into Property-Based frameworks.

Example Properties for longestCommonSubstring

These are some examples of properties for longestCommonSubstring:

for all (x, y)
longestCommonSubstring(x, y) should be equal to longestCommonSubstring(y, x)

for all (x, y)
the result of longestCommonSubstring(x, y) should be present in both x and y

for all (x, y, z)
the result of longestCommonSubstring(y + x + z, x) should be x

Based on the example properties of longestCommonSubstring, do you think you could have come up with these properties? (Please think carefully) *

- ☐ Yes, and I CAN think of a few more.
- ☐ Yes, but I CAN'T think of more.
- ☐ No, they seem hard to come up with.

Example of the implementation of Property-Based Tests for longestCommonSubstring

```
describe('Properties of Longest Common Substring', () => {
  it('should find the same substring lengths whatever the order of the inputs {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), (s1, s2) => {
        assert.equal(longestCommonSubstring(s1, s2).length, longestCommonSubstring(s2, s1).length)
      })
    )
  })

  it('should include the substr in both strings {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), (s1, s2) => {
        const longest = longestCommonSubstring(s1, s2)
        assert.ok(s1.includes(longest))
        assert.ok(s2.includes(longest))
      })
    )
  })

  it('should detect the longest common {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), fc.string(), (s, prefix, suffix) => {
        assert.equal(longestCommonSubstring(prefix + s + suffix, s), s)
      })
    )
  })
})
```

Assuming the Example-Based tests of the previous section for `longestCommonSubstring` were incomplete, if with Property-Based Testing you were able to widen the spectrum of test possibilities and, consequently, find more bugs with the tool would you consider writing Property-Based tests? *

- ☐ Yes, and they seem simpler than having to think of all the test cases by myself.
- ☐ Yes, even if the framework seems a bit complicated at first.
- ☐ No, I would prefer to think of the test cases myself.

Example Properties for myParseInt

These are some examples of properties for `myParseInt`:

for all (x: integer)
where y is the stringified version of x
`myParseInt(y)` should be equal to `parseInt(y)`

for all (x: double)
where y is the stringified version of x
`myParseInt(y)` should be equal to `parseInt(y)`

Based on the example properties of `myParseInt`, do you think you could have come up with these properties? (Please think carefully) *

- ☐ Yes, and I CAN think of a few more.
- ☐ Yes, but I CAN'T think of more.
- ☐ No, they seem hard to come up with.

Example of the implementation of Property-Based Tests for myParseInt

```
describe('Properties of myParseInt', function () {
  it('parsing integers {myParseInt}', () => {
    fc.assert(
      fc.property(fc.integer(), function (num) {
        const string1 = fc.stringify(num)
        expect(myParseInt(string1)).to.equal(parseInt(string1))
      })
    )
  })

  it('parsing doubles {myParseInt}', () => {
    fc.assert(
      fc.property(fc.double(), function (num) {
        const string1 = fc.stringify(num)
        expect(myParseInt(string1)).to.equal(parseInt(string1))
      })
    )
  })
})
```

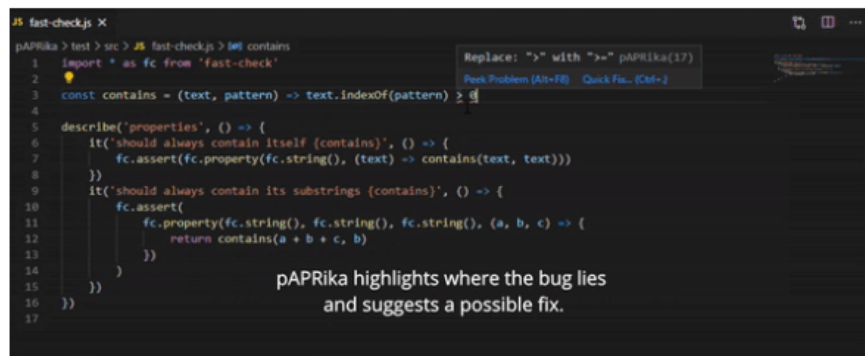
Assuming the Example-Based tests of the previous section for `myParseInt` were incomplete, if with Property-Based Testing you were able to widen the spectrum of test possibilities and, consequently, find more bugs with the tool would you consider writing Property-Based tests? *

- ☐ Yes, and they seem simpler than having to think of all the test cases by myself.
- ☐ Yes, even if the framework seems a bit complicated at first.
- ☐ No, I would prefer to think of the test cases myself.

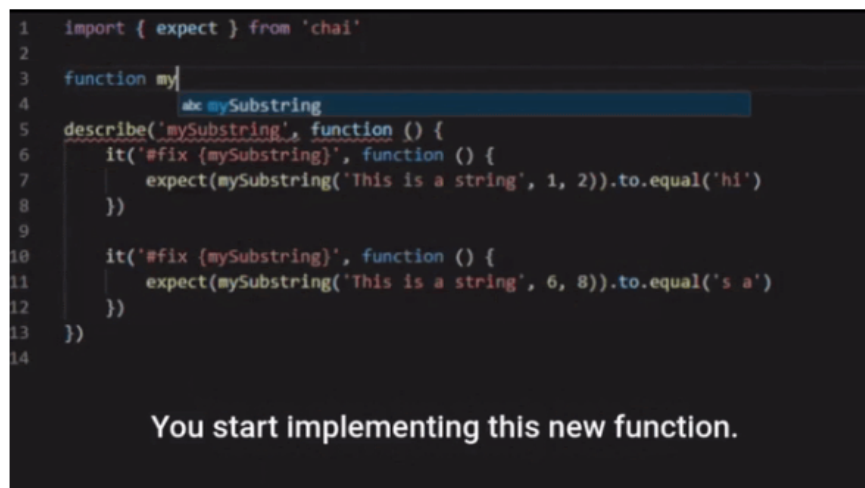
Tool Overview

pAPRika is a prototype extension to Visual Studio Code, repairing JavaScript and TypeScript code and offering suggestions to the developers in real-time, using a mutation-based approach to Automated Program Repair in order to generate patches.

pAPRika Introduction



Why pAPRika might be useful



Based on what you saw on the previous gifs, please answer the following *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The tool's features were simple to use and easy to understand.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
This tool can positively impact my development workflow.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I would consider using this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
A tool like this is likely to distract me from my development.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I preferred the use of the tool On Commands versus On Save	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I would trust the tool	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Based on what you saw on the previous gifs, which features would you say were the most important in pAPRika? *

- ☐ Support for JavaScript/TypeScript.
- ☐ Number of code mutations attempts.
- ☐ Performance.
- ☐ Number of possible fixes.
- ☐ Complexity of suggested fixes.
- ☐ Adaptability to other IDEs.
- ☐ Support for all function types (exported, arrow functions, methods, etc.).

Appendix B

Usability Questionnaire

pAPRika - Automatic Program Repair Extension

This survey aims to evaluate the usability of pAPRika, an IDE extension for JavaScript and TypeScript capable of finding bugs in your code and suggesting small fixes. This extension was developed in the context of a master thesis of the Master in Informatics and Computing Engineering at FEUP called "Property Tests as Specifications Towards Better Code Completion".

All answers are anonymous and will be used exclusively for academic purposes. If you're interested and consent, your participation will be acknowledged in our work.

Thank you for your collaboration,
Afonso Ramos

(Estimated completion time: 4-5 mins)

*Required

Are you participating in this study willingly and do you consent that all opinions will be honest and critical? *

☐ Yes, and I give my consent.

Participant's Profile

This section will only be used to better understand the distributions among the participants. No filling is required if you don't want to share your personal information.

Education *

Highest completed degree of education. If currently enrolled in one, choose the highest one completed.

- ☐ High school
- ☐ Bachelor's degree
- ☐ Master's degree
- ☐ Doctorate's degree

Years of Professional Experience *

Your answer

Technical Background

Please describe your knowledge of software development, JavaScript, debugging, the use of testing frameworks, among others.

Answer according to your background. *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I have considerable experience with JavaScript.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with the Mocha testing framework.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with Visual Studio Code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with Test Driven Development.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I regularly use tools to help me code (linters, code completion, etc.).	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I am always capable of understanding code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I feel comfortable in identifying bugs in code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I feel comfortable in fixing bugs in code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Group *

- ☐ Group A
- ☐ Group B

Group A

Now, in part 1, you should attempt to repair the given problem set WITH the live APR tool resulting from this work.

In part 2, you should attempt to repair the given problem set WITHOUT any tool.

Part 1 *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The bugs were easy to identify.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The solutions were straightforward.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I solved every problem correctly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I spent more time in identifying the bug than in solving it.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The extension was faster in identifying fixes than me.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The extension was able to correctly fix problems.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I used the fixes suggested by the extension.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I tried to understand the fixes suggested by the extension before accepting them.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Part 2 *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The bugs were easy to identify.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The solutions were straightforward.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I solved every problem correctly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I spent more time in identifying the bug than in solving it.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Group B

Now, in part 1, you should attempt to repair the given problem set WITHOUT any tool.

In part 2, you should attempt to repair the given problem set WITH the live APR tool resulting from this work.

Part 1 *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The bugs were easy to identify.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The solutions were straightforward.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I solved every problem correctly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I spent more time in identifying the bug than in solving it.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Part 2 *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The bugs were easy to identify.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The solutions were straightforward.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I solved every problem correctly.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I spent more time in identifying the bug than in solving it.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The extension was faster in identifying fixes than me.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The extension was able to correctly fix problems.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I used the fixes suggested by the extension.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I tried to understand the fixes suggested by the extension before accepting them.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Technical Background

Please describe your knowledge of software development, JavaScript, debugging, the use of testing frameworks, among others.

Answer according to your background. *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
I have considerable experience with JavaScript.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with the Mocha testing framework.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with Visual Studio Code.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I have considerable experience with Test Driven Development.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I regularly use tools to help me code (linters, code completion, etc.).	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I am always capable of understanding code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I feel comfortable in identifying bugs in code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I feel comfortable in fixing bugs in code I haven't seen before.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Example-Based Test Cases

We will now present you with a set of test cases that enable the use of our tool.

With myParseInt one intends to implement a function capable of replicating JavaScript's own parseInt.

With longestCommonSubstring one intends to implement a function capable of returning the longest common substring between two strings.

Example-Based tests for myParseInt

```
describe('myParseInt', function () {
  it('should return same values as JavaScript's parseInt(). #fix {myParseInt} (1)', function () {
    expect(myParseInt('10')).toEqual(parseInt('10'))
  })
  it('should return same values as JavaScript's parseInt(). #fix {myParseInt} (2)', function () {
    expect(myParseInt('00 abs')).toEqual(parseInt('00 abs'))
  })
  it('should return NaN. #fix {myParseInt} (3)', function () {
    expect(myParseInt('abc 95')).toBe.NaN
  })
  it('should return NaN. #fix {myParseInt} (4)', function () {
    expect(myParseInt(' asd45.23')).toBe.NaN
  })
  it('should return same values as JavaScript's parseInt(). #fix {myParseInt} (5)', function () {
    expect(myParseInt('999.4')).toEqual(parseInt('999.4'))
  })
})
```

Do you think the tests above are enough for a developer to replicate JavaScript's parseInt behaviour? *

- ☐ Yes, they seem to cover most of the spectrum of possibilities.
- ☐ No, but I CAN'T think of more different test cases.
- ☐ No, and I CAN think of more tests to incorporate.

Example-Based tests for longestCommonSubstring

```
describe('Longest Common Substring', () => {
  it('should find "sentence" {longestCommonSubstring}', () => {
    assert.equal(
      longestCommonSubstring(
        'This is a sentence that will get analysed a lot',
        'There are a lot of sentences with a lot of common substrings'
      ),
      'sentence'
    )
  })

  it('should find "mportant" {longestCommonSubstring}', () => {
    assert.equal(longestCommonSubstring('wording is important', 'Important wording'), 'mportant')
  })

  it('should find "ika" {longestCommonSubstring}', () => {
    assert.equal(longestCommonSubstring('pAPRIka', 'paprIka'), 'ika')
  })
})
```

Do you think the tests above are enough to always return the longest common substring? *

- ☐ Yes, they seem to cover most of the spectrum of possibilities.
- ☐ No, but I CAN'T think of more different test cases.
- ☐ No, and I CAN think of more tests to incorporate.

Property-Based Testing

Property-based tests are designed to test the aspects of a property that should always be true, rather than having to write a different test for every value that you want to test.

For the questions below please assume that there were missing test cases in the Example-Based tests above.

Defining Properties, a brief tutorial

Property-Based Testing relies on properties. It checks that a function, program or whatever system under test abides by a property. Most of the time, properties do not have to go into specific details about the output. They just have to check for useful characteristics that must be seen in the output.

A property is just something like:

for all (x, y, ...)
such as precondition(x, y, ...)
holds property(x, y, ...) is true

Property example:

for all (a, b, c) strings
the concatenation of a, b and c always contains b

However, these properties still need to be implemented into Property-Based frameworks.

Example Properties for longestCommonSubstring

These are some examples of properties for longestCommonSubstring:

for all (x, y)
longestCommonSubstring(x, y) should be equal to longestCommonSubstring(y, x)

for all (x, y)
the result of longestCommonSubstring(x, y) should be present in both x and y

for all (x, y, z)
the result of longestCommonSubstring(y + x + z, x) should be x

Based on the example properties of longestCommonSubstring, do you think you could have come up with these properties? (Please think carefully) *

- ☐ Yes, and I CAN think of a few more.
- ☐ Yes, but I CAN'T think of more.
- ☐ No, they seem hard to come up with.

Example of the implementation of Property-Based Tests for longestCommonSubstring

```
describe('Properties of longest Common Substring', () => {
  it('should find the same substring lengths whatever the order of the inputs {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), (s1, s2) => {
        assert.equal(longestCommonSubstring(s1, s2).length, longestCommonSubstring(s2, s1).length)
      })
    )
  })

  it('should include the substr in both strings {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), (s1, s2) => {
        const longest = longestCommonSubstring(s1, s2)
        assert.ok(s1.includes(longest))
        assert.ok(s2.includes(longest))
      })
    )
  })

  it('should detect the longest common {longestCommonSubstring}', () => {
    fc.assert(
      fc.property(fc.string(), fc.string(), fc.string(), (s, prefix, suffix) => {
        assert.equal(longestCommonSubstring(prefix + s + suffix, s), s)
      })
    )
  })
})
```

Assuming the Example-Based tests of the previous section for `longestCommonSubstring` were incomplete, if with Property-Based Testing you were able to widen the spectrum of test possibilities and, consequently, find more bugs with the tool would you consider writing Property-Based tests? *

- ☐ Yes, and they seem simpler than having to think of all the test cases by myself.
- ☐ Yes, even if the framework seems a bit complicated at first.
- ☐ No, I would prefer to think of the test cases myself.

Example Properties for `myParseInt`

These are some examples of properties for `myParseInt`:

for all (x: integer)
where y is the stringified version of x
`myParseInt(y)` should be equal to `parseInt(y)`

for all (x: double)
where y is the stringified version of x
`myParseInt(y)` should be equal to `parseInt(y)`

Based on the example properties of `myParseInt`, do you think you could have come up with these properties? (Please think carefully) *

- ☐ Yes, and I CAN think of a few more.
- ☐ Yes, but I CAN'T think of more.
- ☐ No, they seem hard to come up with.

Example of the implementation of Property-Based Tests for `myParseInt`

```
describe('Properties of myParseInt', function () {
  it('parsing integers {myParseInt}', () => {
    fc.assert(
      fc.property(fc.integer(), function (num) {
        const string1 = fc.stringify(num)
        expect(myParseInt(string1)).to.equal(parseInt(string1))
      })
    )
  })

  it('parsing doubles {myParseInt}', () => {
    fc.assert(
      fc.property(fc.double(), function (num) {
        const string1 = fc.stringify(num)
        expect(myParseInt(string1)).to.equal(parseInt(string1))
      })
    )
  })
})
```

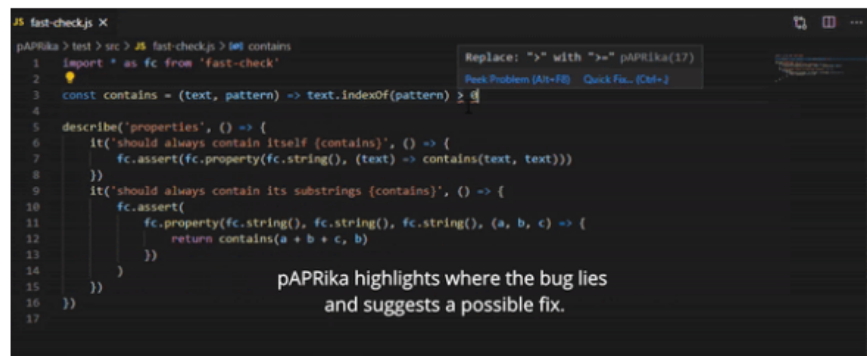
Assuming the Example-Based tests of the previous section for `myParseInt` were incomplete, if with Property-Based Testing you were able to widen the spectrum of test possibilities and, consequently, find more bugs with the tool would you consider writing Property-Based tests? *

- ☐ Yes, and they seem simpler than having to think of all the test cases by myself.
- ☐ Yes, even if the framework seems a bit complicated at first.
- ☐ No, I would prefer to think of the test cases myself.

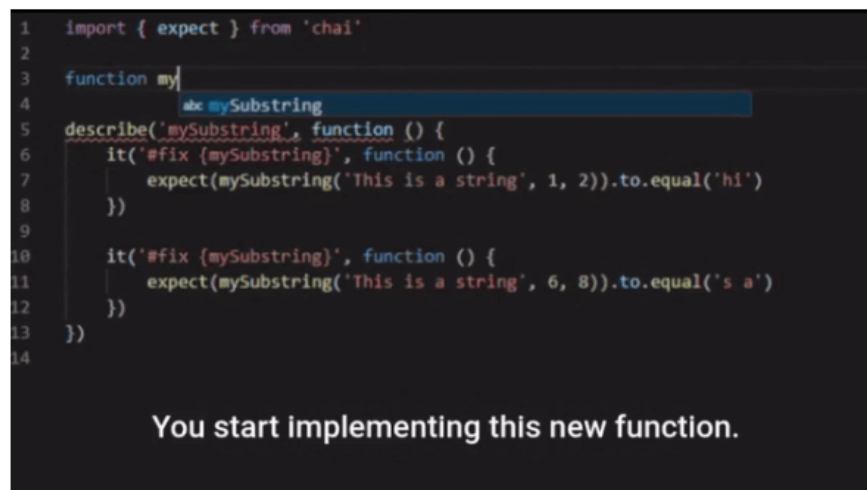
Tool Overview

pAPRika is a prototype extension to Visual Studio Code, repairing JavaScript and TypeScript code and offering suggestions to the developers in real-time, using a mutation-based approach to Automated Program Repair in order to generate patches.

pAPRika Introduction



Why pAPRika might be useful



Based on your experience with the tool, please answer the following *

	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
The tool's features were simple to use and easy to understand.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
This tool can positively impact my development workflow.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I would consider using this tool.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
A tool like this is likely to distract me from my development.	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I preferred the use of the tool On Commands versus On Save	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
I would trust the tool	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

Based on your experience with the tool, which features would you say were the most important in pAPRika? *

- ☐ Support for JavaScript/TypeScript.
- ☐ Number of code mutations attempts.
- ☐ Performance.
- ☐ Number of possible fixes.
- ☐ Complexity of suggested fixes.
- ☐ Adaptability to other IDEs.
- ☐ Support for all function types (exported, arrow functions, methods, etc.).

Based on your experience with the tool, which features could be improved? *

- ☐ Support for JavaScript/TypeScript.
- ☐ Number of code mutations attempts.
- ☐ Performance.
- ☐ Number of possible fixes.
- ☐ Complexity of suggested fixes.
- ☐ Adaptability to other IDEs.
- ☐ Support for all function types (exported, arrow functions, methods, etc.).

References

- [AC05] Christine M Anderson-Cook. Experimental and Quasi-Experimental Designs for Generalized Causal Inference. *Journal of the American Statistical Association*, 100(470):708–708, 2005. Cited on p. 69.
- [AHM06] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings - International Conference on Software Engineering*, volume 2006, pages 361–370, 2006. Cited on p. 11.
- [Alb97] Gerald Albaum. The Likert Scale Revisited. *Market Research Society. Journal.*, 39(2):1–21, mar 1997. Cited on p. 55.
- [ARC⁺19] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: Tightening the feedback loops. In *ACM International Conference Proceeding Series*, pages 1–6, New York, New York, USA, apr 2019. ACM Press. Cited on p. 10.
- [AWG⁺20] Gene M. Alarcon, Charles Walter, Anthony M. Gibson, Rose F. Gamble, August Capiola, Sarah A. Jessup, and Tyler J. Ryan. Would You Fix This Code for Me? Effects of Repair Source and Commenting on Trust in Code Repair. *Systems*, 8(1):8, mar 2020. Cited on p. 78.
- [AZvG08] Rui Abreu, Peter Zoetewij, and Arjan J.C. van Gemund. On the Accuracy of Spectrum-based Fault Localization. pages 89–98. Institute of Electrical and Electronics Engineers (IEEE), apr 2008. Cited on p. 14.
- [BB19] J. A. Bergstra and M. Burgess. A Promise Theoretic Account of the Boeing 737 Max MCAS Algorithm Affair. dec 2019. Cited on p. 11.
- [BF74] Morton B. Brown and Alan B. Forsythe. Robust tests for the equality of variances. *Journal of the American Statistical Association*, 69(346):364–367, 1974. Cited on p. 56.
- [BJ07] Paul Brook and Daniel Jacobowitz. Reversible debugging. *Proceedings of the GCC Developers’ Summit*, pages 69–76, 2007. Cited on p. 2.
- [BMM09] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *ESEC-FSE’09 - Proceedings of the Joint 12th European Software Engineering Conference and 17th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 213–222, 2009. Cited on pp. 19 and 21.

- [BS14] Richard E. Bourque, Pierre and Fairley and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. IEEE Computer Society Press, 3rd edition, 2014. Cited on p. 8.
- [BYP19] Rohan Bavishi, Hiroaki Yoshida, and Mukul R. Prasad. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 613–624, New York, New York, USA, aug 2019. ACM Press. Cited on p. 79.
- [Cam19] Diogo Campos. *Tests as Specifications towards better Code Completion*. PhD thesis, Faculty of Engineering of the University of Porto, 2019. Cited on pp. 2, 19, 22, 23, 25, 27, 40, 45, 46, 48, 52, 54, 55, 56, 59, 70, and 74.
- [CH00] Koen Claessen and John Hughes. QuickCheck. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming - ICFP '00*, pages 268–279, New York, New York, USA, 2000. ACM Press. Cited on pp. 17 and 41.
- [CPF17] Liushan Chen, Yu Pei, and Carlo A Furia. Contract-based program repair without the contracts. In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 637–647, 2017. Cited on p. 79.
- [CSC⁺19] Jose Pablo Cambronero, Jiasi Shen, Jurgen Cito, Elena Glassman, and Martin Rinard. Characterizing Developer Use of Automatically Generated Patches. In *2019 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, volume 2019-Octob, pages 181–185. IEEE, oct 2019. Cited on pp. 21 and 75.
- [DLTL19] Zhen Yu Ding, Yiwei Lyu, Christopher Timperley, and Claire Le Goues. Leveraging Program Invariants to Promote Population Diversity in Search-Based Automatic Program Repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*, pages 2–9. IEEE, may 2019. Cited on pp. 12 and 29.
- [DW10] Vidroha Debroy and W. Eric Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation*, pages 65–74. IEEE, 2010. Cited on pp. 28 and 31.
- [dW13] J. C.F. de Winter. Using the student’s t-test with extremely small sample sizes. *Practical Assessment, Research and Evaluation*, 18(10):1–12, 2013. Cited on p. 56.
- [EKS93] Shari Ellis, David Klahr, and Robert S. Siegler. Effects of feedback and collaboration on changes in children’s use of mathematical rules. *Meetings of the Society for Research in Child Development*, 1993. Cited on pp. 1 and 9.
- [FB97] George Fink and Matt Bishop. Property-Based Testing ; A New Approach to Testing for Assurance. *Software Engineering Notes*, 22(4):74, 1997. Cited on p. 17.
- [FD14] Alfonso Fuggetta and Elisabetta Di Nitto. Software process. In *Proceedings of the on Future of Software Engineering - FOSE 2014*, pages 1–12, New York, New York, USA, may 2014. ACM Press. Cited on pp. 1 and 9.
- [FL94] George Fink and Karl Levitt. Property-based testing of privileged programs. In *Annual Computer Security Applications Conference*, pages 154–163, 1994. Cited on p. 17.

- [FNWL09] Stephanie Forrest, Thanhvu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Genetic and Evolutionary Computation Conference, GECCO-2009*, pages 947–954, 2009. Cited on p. 11.
- [FPG94] Norman Fenton, S.L. Pfleeger, and R.L. Glass. Science and substance: a challenge to software engineers. *IEEE Software*, 11(4):86–95, jul 1994. Cited on pp. 23 and 46.
- [Fra93] Frankreich Convention Nationale. *Collection générale des décrets rendus par la Convention Nationale*. Chez Baudouin, Imprimeur de la Convention Nationale. A, Paris., 1793. Cited on p. 1.
- [GGM19] Negar Ghorbani, Joshua Garcia, and Sam Malek. Detection and Repair of Architectural Inconsistencies in Java. In *Proceedings - International Conference on Software Engineering*, volume 2019-May, pages 560–571. IEEE Computer Society, may 2019. Cited on p. 20.
- [Gin19] Davide Ginelli. Failure-driven program repair. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2019*, pages 1156–1159, New York, New York, USA, 2019. ACM Press. Cited on p. 13.
- [GMM19] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, jan 2019. Cited on pp. 12, 14, 15, 16, and 19.
- [GPR19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, nov 2019. Cited on pp. 13 and 20.
- [Häh13] Reiner Hähnle. The abstract behavioral specification language: A tutorial introduction. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7866 LNCS, pages 1–37. Springer, Berlin, Heidelberg, 2013. Cited on p. 13.
- [HAM⁺20] Yang Hu, Umair Z. Ahmed, Sergey Mehtaev, Ben Leong, and Abhik Roychoudhury. Re-Factoring Based Program Repair Applied to Programming Assignments. pages 388–398. Institute of Electrical and Electronics Engineers (IEEE), jan 2020. Cited on p. 12.
- [HO18] Mark Harman and Peter O’Hearn. From Start-ups to Scale-ups: Opportunities and Open Problems for Static and Dynamic Program Analysis. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 1–23. IEEE, sep 2018. Cited on p. 20.
- [Hus16] Husson University- Online. What is the Software Development Cycle?, 2016. Cited on p. 9.
- [JAV09] Tom Janssem, Rui Abreu, and Arjan J.C. Van Gemund. Zoltar: A toolset for automatic fault localization. In *ASE2009 - 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664, 2009. Cited on p. 14.

- [JH05] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, pages 273–282, 2005. Cited on pp. 11 and 14.
- [JJE14] René Just, Darioush Jalali, and Michael D Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings*, pages 437–440, 2014. Cited on pp. 31 and 32.
- [JXZ⁺18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. Shaping program repair space with existing patches and similar code. In *ISSTA 2018 - Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, volume 18, pages 298–309. ACM, 2018. Cited on pp. 12 and 13.
- [KA72] Raymond W. Kulhavy and Richard C. Anderson. Delay-retention effect with multiple-choice tests. *Journal of Educational Psychology*, 63(5):505–512, 1972. Cited on pp. 1 and 9.
- [KHL11] Damir Kalpić, Nikica Hlupić, and Miodrag Lovrić. Student’s t-Tests. In *International Encyclopedia of Statistical Science*, pages 1559–1563. Springer Berlin Heidelberg, 2011. Cited on p. 56.
- [KKH13] Paul Kehrer, Kim Kelly, and Neil Heffernan. Does immediate feedback while doing homework improve learning? In *FLAIRS 2013 - Proceedings of the 26th International Florida Artificial Intelligence Research Society Conference*, pages 542–545, 2013. Cited on pp. 1 and 9.
- [KLB⁺19] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. IFixR: Bug report driven program repair. In *ESEC/FSE 2019 - Proceedings of the 2019 27th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 314–325. Association for Computing Machinery, Inc, aug 2019. Cited on pp. 21 and 79.
- [KNSK13] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *Proceedings - International Conference on Software Engineering*, pages 802–811, 2013. Cited on p. 12.
- [Koz10] John R. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3-4):251–284, sep 2010. Cited on p. 11.
- [LB12] Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '12*, page 133, New York, New York, USA, 2012. ACM Press. Cited on pp. 12, 13, and 19.
- [LCL⁺17] Xuan Bach D. Le, Duc Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume Part F1301, pages 593–604, 2017. Cited on p. 21.

- [LDVFW12] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each. In *Proceedings - International Conference on Software Engineering*, pages 3–13, 2012. Cited on p. 12.
- [Le16] Xuan-Bach D Le. Towards efficient and effective automatic program repair. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, pages 876–879, New York, New York, USA, 2016. ACM Press. Cited on pp. 16 and 17.
- [Leh80] Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. Cited on pp. 2 and 17.
- [LFW13] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, sep 2013. Cited on pp. 12, 20, and 21.
- [Lik32] R Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22 140:55, 1932. Cited on pp. 48, 49, and 55.
- [LKK⁺18] Kui Liu, Anil Koyuncu, Kisub Kim, Dongsun Kim, and Tegawende F. Bissyande. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, volume 2018-Decem, pages 658–662. IEEE Computer Society, jul 2018. Cited on pp. 12 and 13.
- [LKKB19] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F Bissyandé. TBAR: Re-visiting template-based automated program repair. In *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 43–54. ACM, 2019. Cited on pp. 12 and 13.
- [LLG16] Xuan Bach D. Le, David Lo, and Claire Le Goues. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 213–224. IEEE, mar 2016. Cited on p. 12.
- [LLL16] Xuan-Bach D. Le, David Lo, and Claire Le Goues. Empirical Study on Synthesis Engines for Semantics-Based Program Repair. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 423–427. IEEE, oct 2016. Cited on p. 20.
- [LNF12] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, jan 2012. Cited on pp. 12, 13, and 29.
- [LNZ⁺05] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *ACM SIGPLAN Notices*, 40(6):15, jun 2005. Cited on p. 11.
- [LR01] Meir M. Lehman and Juan F. Ramil. Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering*, 11(1):15–44, 2001. Cited on pp. 2 and 17.
- [LR15] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings*, pages 166–178, 2015. Cited on p. 12.

- [LR16a] Fan Long and Martin Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings - International Conference on Software Engineering*, volume 14-22-May-, pages 702–713, 2016. Cited on pp. 16 and 20.
- [LR16b] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*, pages 298–312, New York, New York, USA, 2016. ACM Press. Cited on p. 12.
- [LTA11] Huiqing Li, Simon Thompson, and Thomas Arts. Extracting properties from test cases by refactoring. In *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, pages 472–473, 2011. Cited on p. 18.
- [MBC⁺19] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. SapFix: Automated End-to-End Repair at Scale. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 269–278. IEEE, may 2019. Cited on pp. 12, 23, 29, 40, and 69.
- [MM18] Matias Martinez and Martin Monperrus. Astor: Exploring the Design Space of Generate-and-Validate Program Repair beyond GenProg. *Journal of Systems and Software*, 151:65–80, feb 2018. Cited on pp. 12, 21, and 29.
- [Mon18] Martin Monperrus. Automatic Software Repair: a Bibliography. *ACM Computing Surveys*, 51(1):1–24, jul 2018. Cited on pp. 12 and 13.
- [MS06] Erica Mealy and Paul Strooper. Evaluating software refactoring tool support. In *Australian Software Engineering Conference (ASWEC’06)*, volume 2006, pages 10 pp.–340. IEEE, 2006. Cited on pp. 2, 3, 8, and 11.
- [MYR15] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 448–458. IEEE, may 2015. Cited on p. 21.
- [MYR16] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In *Proceedings - International Conference on Software Engineering*, volume 14-22-May-, pages 691–701, 2016. Cited on pp. 12, 13, and 20.
- [OS19] Michael O’Neill and Lee Spector. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, sep 2019. Cited on pp. 11, 20, and 77.
- [OVGB10] Michael O’Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. Open issues in Genetic Programming. *Genetic Programming and Evolvable Machines*, 11(3-4):339–363, sep 2010. Cited on p. 10.
- [PFNM15] Yu Pei, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automated Program Repair in an Integrated Development Environment. In *Proceedings - International Conference on Software Engineering*, volume 2, pages 681–684. IEEE Computer Society, aug 2015. Cited on p. 19.

- [PZ14] Mauro Pezzè and Cheng Zhang. Automated test oracles: A survey. In *Advances in Computers*, volume 95, pages 1–48. Academic Press Inc., 2014. Cited on p. 12.
- [QLAR15] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *2015 International Symposium on Software Testing and Analysis, ISSTA 2015 - Proceedings*, pages 24–36, 2015. Cited on p. 20.
- [QML⁺14] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziyang Dai, and Chengsong Wang. The strength of random search on automated program repair. In *Proceedings - International Conference on Software Engineering*, number 1, pages 254–265, 2014. Cited on p. 21.
- [RAW⁺19] Tyler J. Ryan, Gene M. Alarcon, Charles Walter, Rose Gamble, Sarah A. Jessup, August Capiola, and Marc D. Pfahler. Trust in Automated Software Repair: The Effects of Repair Source, Transparency, and Programmer Experience on Perceived Trustworthiness and Trust. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11594 LNCS, pages 452–470. Springer Verlag, jul 2019. Cited on p. 78.
- [REN19] Portugueses são dos que menos conseguem conciliar trabalho e universidade - Renascença, 2019. Cited on p. 64.
- [RW88] Charles Rich and R.C. Waters. Automatic programming: myths and prospects. *Computer*, 21(8):40–51, aug 1988. Cited on p. 11.
- [Sam59] A L Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3(3):210–229, jul 1959. Cited on p. 10.
- [SBLB15] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 532–543, New York, New York, USA, 2015. ACM Press. Cited on pp. 2, 20, and 21.
- [SDM⁺18] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 2018-March, pages 130–140. IEEE, mar 2018. Cited on p. 32.
- [SL18] Mauricio Soto and Claire Le Goues. Using a probabilistic model to predict bug fixes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, volume 2018-March, pages 221–231. IEEE, mar 2018. Cited on p. 21.
- [SMJ15] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. Are students representatives of professionals in software engineering experiments? In *Proceedings - International Conference on Software Engineering*, volume 1, pages 666–676. IEEE Computer Society, aug 2015. Cited on p. 71.

- [SND⁺11] Diptikalyan Saha, Mangala Gowri Nanda, Pankaj Dhoolia, V. Krishna Nandivada, Vibha Sinha, and Satish Chandra. Fault localization for data-centric programs. In *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 157–167, New York, New York, USA, 2011. ACM Press. Cited on p. 11.
- [Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2010. Cited on pp. 7 and 8.
- [SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley Professional, 2003. Cited on pp. 3 and 11.
- [SSP19] Seemanta Saha, Ripon K Saha, and Mukul R Prasad. Harnessing Evolution for Multi-Hunk Program Repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, volume 2019-May, pages 13–24. IEEE, may 2019. Cited on pp. 12 and 14.
- [SW65] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591, dec 1965. Cited on p. 56.
- [SWH11] Matt Staats, Michael W Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles: The foundations of testing revisited. In *Proceedings - International Conference on Software Engineering*, number 11, pages 391–400, 2011. Cited on p. 14.
- [Tan90] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, jun 1990. Cited on p. 9.
- [Tan13] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, may 2013. Cited on pp. 2, 3, 9, 10, 19, 21, and 79.
- [TPW⁺18] Michele Tufano, Massimiliano Di Penta, Cody Watson, Martin White, Gabriele Bavota, and Denys Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *ASE 2018 - Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 832–837, 2018. Cited on pp. 12, 21, and 43.
- [VAH18] Hoang Van Thuy, Phan Viet Anh, and Nguyen Xuan Hoai. Automated Large Program Repair based on Big Code. In *Proceedings of the Ninth International Symposium on Information and Communication Technology - SoICT 2018*, pages 375–381, New York, New York, USA, 2018. ACM Press. Cited on p. 12.
- [War20] Charlie Warzel. Opinion | The App That Broke the Iowa Caucus - The New York Times, feb 2020. Cited on p. 8.
- [WNLF09] Westley Weimer, Thanh Vu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings - International Conference on Software Engineering*, pages 364–374, 2009. Cited on pp. 12 and 29.
- [WRH⁺12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, Anders Wesslén, Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. Empirical Strategies. In *Experimentation in Software Engineering*, pages 9–36. Springer Berlin Heidelberg, 2012. Cited on pp. 69 and 70.

- [XLZ⁺18] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*, volume 11, pages 789–799, New York, New York, USA, 2018. ACM Press. Cited on pp. 12 and 13.
- [XMD⁺17] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, 43(1):34–55, jan 2017. Cited on pp. 12 and 13.
- [YB18] Yuan Yuan and Wolfgang Banzhaf. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, dec 2018. Cited on p. 21.
- [YYZ⁺11] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairava-sundaram. How do fixes become bugs? A comprehensive characteristic study on incorrect fixes in commercial and open source operating systems. In *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 26–36, 2011. Cited on p. 11.
- [ZC09] Michael Zhivich and Robert K. Cunningham. The real cost of software errors. *IEEE Security and Privacy*, 7(2):87–90, mar 2009. Cited on p. 11.
- [ZS15] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In *Proceedings - International Conference on Software Engineering*, volume 1, pages 913–923, 2015. Cited on p. 12.
- [ZZH⁺19] Jie M. Zhang, Lingming Zhang, Dan Hao, Lu Zhang, and Mark Harman. An empirical comparison of mutant selection assessment metrics. In *Proceedings - 2019 IEEE 12th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2019*, pages 90–101. Institute of Electrical and Electronics Engineers Inc., apr 2019. Cited on p. 21.