FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Automated Generic Optimization in Real Time using Mutation Operators

Pedro Miguel Oliveira Carvalho da Silva

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Monteiro de Oliveira Restivo, Assistant Professor

Second Supervisor: Hugo José Sereno Lopes Ferreira, Assistant Professor

July 19, 2021

Automated Generic Optimization in Real Time using Mutation Operators

Pedro Miguel Oliveira Carvalho da Silva

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Prof. Tiago Boldt Sousa
External Examiner: Prof. João Saraiva
Supervisor: Prof. André Restivo
Co-Supervisor: Prof. Hugo S. Ferreira

July 19, 2021

Abstract

Developing and maintaining a software project can be a lengthy and costly endeavor, especially on bigger scales. However, and most prominently in schedule-driven environments, code efficiency, energy costs, and other performance metrics are bound to be compromised to achieve the goals and deliver within the deadlines. These compromises can have a serious impact on the quality and health of the codebase and even the viability of the product in the future due to slow executions or excessive energy costs.

Automated Programming (AP), by definition, is a type of computer programming that allows a human to tell a computer what to do in a progressively more abstract way. Ultimately, it aims to reduce the input from the developer, leading to a more automatized process. However, some predict that in the future, automatic programming will be more about fixing mistakes rather than having an autonomous machine do the work.

Automatic Programming Repair (APR) is a research area that aims to find and fix bugs. This goal is generally accomplished by finding mutations of a segment of a program that fix what is considered to be wrong based on an empirical source of truth, typically Unit Tests (UT) and Property-Based Tests (PBT).

In this work, we propose a framework that is able to mutate programs with mutation operators to find variant programs that are more optimal to a user-defined metric. The framework would then suggest to the user, in real-time, how to optimize their code in a metric like *speed*, *energy*, and *program correctness* (program repair). This will shorten the time to reach a patch during development and thus shortening the length of development and consequently, its costs.

By implementing the described framework as a *Visual Studio Code* extension, and conducting an empirical study with 14 participants, we conclude that there is significant evidence that participants when using our tool, are significantly faster to reach a patch. Furthermore, we found that there are some differences in code written by developers and the patches suggested by our tool. Moreover, we established that users are not always critical when accepting the tool's suggestions which may lead to the code base becoming harder to interpret for developers when the suggestion is not what it would be expected when translating the code to natural language.

Resumo

Desenvolver e fazer manutenção de projetos de software é uma tarefa morosa e com custos altos, especialmente em grande escala. Contudo, e mais evidente em ambientes guiados a objetivos, a eficiência do código, o seu custo energético e outras métricas estão destinadas a serem comprometidas em detrimento dos objetivos serem alcançados. Estes compromissos têm um sério impacto na qualidade e saúde do projeto e, consequentemente, até da sua viabilidade futura por possíveis tempos de execução ou custos energéticos demasiado elevados.

Programação Automática (AP), por definição, é um tipo de programação que permite um ser humano explicar a um computador o que fazer de uma forma progressivamente mais abstrata. Em última análise, visa reduzir o input do programador, levando a um processo mais automatizado. Contudo, alguns prevêem que, no futuro, a programação automática será mais sobre como corrigir erros em vez de ter uma máquina autônoma que saiba programar.

Programação de Reparo Automático (APR) é uma área de pesquisa que visa encontrar e corrigir bugs. Este objetivo é geralmente alcançado encontrando mutações de um segmento de um programa que corrige o que é considerado errado com base numa fonte empírica de verdade, normalmente através de testes unitários (UT) de testes baseados em propriedades (PBT).

Neste trabalho, propomos uma framework que é capaz de transformar um programa, através de mutações, em variantes com melhores propriedades de acordo com a métrica definida pelo utilizador. A framework depois sugeriria ao utilizador, em tempo real, como otimizar o seu código mediante uma métrica como *velocidade de execução*, *custo energético* e *correção do programa*. Este processo encurtará o tempo para chegar a um patch durante o desenvolvimento e, assim, diminuirá a duração do desenvolvimento e, consequentemente, o seu custo.

Implementando a framework descrita como uma extensão para o *Visual Studio Code*, e conduzindo um estudo empírico com 14 participantes, concluímos que há indícios significativos de que os participantes, ao usar a nossa ferramenta, foram significativamente mais rápidos para chegar a um patch. Além disso, descobrimos que existem algumas diferenças no código escrito por programadores e nos patches sugeridos pela nossa ferramenta. Adicionalmente, estabelecemos que os usuários nem sempre são críticos ao aceitar as sugestões da ferramenta, o que pode fazer com que o código se torne mais difícil de interpretar para futuros programadores quando a sugestão aceite não é a que seria de se esperar ao traduzir o código para linguagem natural.

Acknowledgements

The completion of this work would not have been possible without the expertise of my supervisors, Professors André Restivo and Hugo Sereno Ferreira.

I would also like to thank all my friends, for all the pointless never-ending arguments for nothing other than the intellectual challenge. For all the moments of fun, for all the early mornings and late nights of work. And especially for the afternoons solving chess puzzles and quizzes when a break from work was needed.

To Catarina, my girlfriend, for the unconditional support, kindness, patience, and love, who, I think unbeknownst to her, has taught me a great deal about life.

And last, but definitely not the least, to my family, there are no words to describe the gratitude I have for all the support throughout my life, I'll be forever grateful — Thank you.

Pedro Silva

"Some people want it to happen,
some wish it would happen,
others make it happen."

Michael Jordan

Contents

List of Figures

List of Tables

Abbreviations

AST	Abstract syntax tree
AP	Automated Programming
APO	Automated Program Optimization
APR	Automated Program Repair
CFG	Control-Flow Graph
CPU	Central Processing Unit
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
LHS	Left-Hand Side
LTS	Long-Term Support
MCCCC	Model-and-Code Consistency Checking
PBT	Property-Based Testing
RAPL	Running Average Power Limit
RHS	Right-Hand Side
UT	Unit Testing
VSCode	Visual Studio Code

Chapter 1

Introduction

This chapter describes the context, problem definition and goals, motivation, and the structure of the document. In Section 1.1 we contextualize our work, in Section 1.3 (p. 2) we explain our problem and what we wish to achieve with this work. In Section 1.2 (p. 2), we elaborate on our motivation to pursue this work and finally, in Section 1.5 (p. 3) we give a brief overview of the document's structure.

1.1 Context

Software development is an ever-growing area with great promise and already great results. However, its production and maintenance is limited by the amount and quality of the developers which can be a very rare and expensive commodity.

With that in mind, a research field by the name of Automated Programming (AP) gained traction. Automated Programming involves the process of translation between the human language and the language of machines [8]. In other words, with AP, the higher-level, human notation, ideally simpler and quicker to write, is automatically translated to some conventional programming language and thus shortens development life cycles and reduces overall costs in software development.

A specific sub-area of research of AP is Automatic Programming Repair (APR), which aims to identify, locate and fix bugs in code [27] by creating patches of code which by themselves may improve the quality of the code, but it does not necessarily optimize its metrics like speed and energy costs.

Furthermore, and tightly connected to AP and APR, there is Live Programming [51], a research area which focuses on how development environment can be smarter than traditional text editors, and can provide valuable information to the developer in real-time about the program's state and other properties. This tight feedback loop between the developer and the live environment is thought to lead to better results faster, especially in cases where *exploration works better than just forward engineering* [4].

1.2 Motivation

A survey conducted by *Jones* [3] reported that the number of software maintenance professionals between the 1950s and the expected value for 2025 would drastically increase from 10% to 77%. Software maintenance includes the addition of new features, optimization of old features, correction of errors, etc., and is reported to consume a major portion of the total cost of a software project [32]. However, in this fast passed environment, the quality of the software is not always the main priority and this can have dire consequences in later stages of development.

Facing this glaring problem, strides towards reducing software development costs have been made in the areas of Automatic Programming and Automated Program Repair to automate the process of developing code, finding and fixing bugs, and maintaining software.

If we could make significant code improvements without or minimizing the intervention of the developers, not only the costs of the improvement would be significantly lower, but the consequences of the new patches (i.e. more energy-friendly patch which reduces energy consumption drastically) would also improve on the given metric having further benefits in the long term.

1.3 Problem Definition

Present-day real-time software development environments are mainly focused on solving a specific problem or small set of problems as it will be further analyzed in Chapter 2 (p. 5) because of the specific techniques used which are meant to best adapt to the goals of each projects.

In development tools like linters and code completion, developers are able to improve not only the quality of the code, but also the speed in which they produce it. However, and like previously mentioned, these are used for very specific applications such as syntax coherence.

In this dissertation, we aim to explore how an Automatic Programming Optimization tool can be implemented in such a way that is able to be extended to support any generic optimization metric and thus empower the developer to choose, in real-time which metric they value the most in their software for each specific program.

1.4 General Goals

Considering that Automated Program Repair is mostly the act of optimizing a program correctness, we aim to create an Automated Program Optimization (APO) tool that can take advantage of the

APR research on mutation operators. Combining it with a flexible evaluation system that allows the tool to evaluate a program not only on its correctness but in any wanted metric, we get an APO tool that is able to mutate programs and search, in the mutation space, for mutation with better metrics.

To reach these goals, we set ourselves up to extend an APR tool originally created by *Campos* [11], and later enhanced by *Ramos* [41] with the intent to generalize how a metric is recorded and analyzed to be able to optimize any generic metric while still using some of the implemented APR techniques. Since it's not feasible to implement the infinite set of generic metrics one can think of, we decided to focus on showing how our tool can implement three different metrics: (1) *program correctness*, meaning does the program run the tests successfully, (2) *program performance*, meaning the execution time of the program and finally, (3) *power consumption*, meaning the energy spent while running the program. These metrics were chosen due to their relevance in the industry due to their associated costs as well as their inherent differences: metric (1) is related to code semantics, metric (2) is related to how fast the program runs in comparison to the machine's system time and finally, metric (3) relates to the energy spent by the hardware of the machine when running the program.

1.5 Document Structure

This document is divided into five chapters:

- Chapter 1 (p. 1), **Introduction**, which introduces the topic, the hypothesis, the goal and the validation method;
- Chapter 2 (p. 5), **State of the Art** describes the current state of the art in automated program repair strategies and performance bottleneck localization;
- Chapter 3 (p. 17), **Problem Statement** presents the solution we propose to the problem and our plan to achieve it;
- Chapter 4 (p. 21), **Framework Implementation**, defaults the implementation of the Automatic Program Optimization tool developed for this work;
- Chapter 5 (p. 31), **Empirical Evaluation**, explains the process and results of the empirical tests undertaken;
- Chapter 6 (p. 51) **Conclusions**, which summarizes the findings and describes ho the future work will be guided.

Chapter 2

State of the Art

This chapter describes the methodology used in our literature review in Section 2.1. In Section 2.2 (p. 6), we give a brief background information about Automatic Programming. Afterward, in Section 2.3 (p. 6), we explore the techniques used in APR that may be of use when developing an APO tool. In Section 2.4 (p. 9), we study localization strategies that may be implemented to allow for the detection of the most expensive code segments in a program. We analyze some Live Programming tools in Section 2.5 (p. 10) to understand how Live Programming tools work and what their core techniques are and finally, in Section 2.6 (p. 15), we summary the contents of this chapter.

2.1 Methodology

To collect information about the state of the art of the proposed topics an iterative approach was followed. An initial set of keywords was defined based on the perceived domain. These keywords were used to guide the search through various databases such as IEEE Xplore, ACM Digital Library, and Scopus.

To determine if an article was relevant, we analyzed its title, abstract, and conclusion individually. To expand the search as much as possible and make sure no synonyms or related topics were missed, new-found terms and related articles were used as a way to find new articles and

branch to new topics. This strategy was loosely based on a bottom-up approach in which the literature results obtained from search queries are complemented with *snowballing* [55], a method that explores related work to the found literature by analyzing the references of each result.

1. Search for a set of keywords in the title, abstract, and keywords in the Scopus database;
2. Refine the queries by including synonyms and removing keywords with a high intersection with unwanted domains;
3. Manually analyze the documents for domain similarity based on abstract analyzes;
4. Use the newfound keywords and articles to repeat the process.

2.2 Automatic Programming

Automatic programming is an ever-changing concept that has evolved throughout time. In early times, compute scientists used this term to refer to *the ability to automatically generate machine code from an assembly language* [37] and similarly with regards to the creation of compilers [34].

It later developed into a higher-level concept which described automatic programming simply as another layer of abstraction where a developer would describe almost in natural language the specifications of the program and the computer would be able to generate it. This would be famously stated by Arthur Samuel: *"tell the machine what to do, not how to do it"* [43].

2.3 Automated Program Repair

Software projects nowadays have strict deadlines and release requirements that impose short bug fixing and maintenance cycles, putting significant pressure on the developers [18]. Automatic program repair proposes solving that mistake by fixing these bugs automatically.

The process is mostly divided into three steps: locate where the bug is (1), generate potential patches (2), and finally, make sure the suggested patch is good enough (3) to grant a suggestion or even an automatic substitution.

Automated program repair aims to find human-made, non-malicious bugs [6] in programs and through mutations [5, 38] find new programs which are similar to the original one without the bug and still having the initial intended behavior [19]. In this section, we study what state-of-the-art APR techniques exist and which may better suit our application.

2.3.1 Generate and Validate

Generate and validate, as seen in Figure 2.1, repairs are generally comprised of three main steps: fault localization, candidate patch generation, and patch selection, and validation. In the first step, the approach locates which are the program sections that are more likely to have bugs based on passing and failing test-cases. In the second step, mutations are created from the original program

to create alternatives that might correct the bugs described in the test suite. Finally, the mutation is tested against the test-suite to make sure the bug was solved and that no other program functionality was compromised [41].

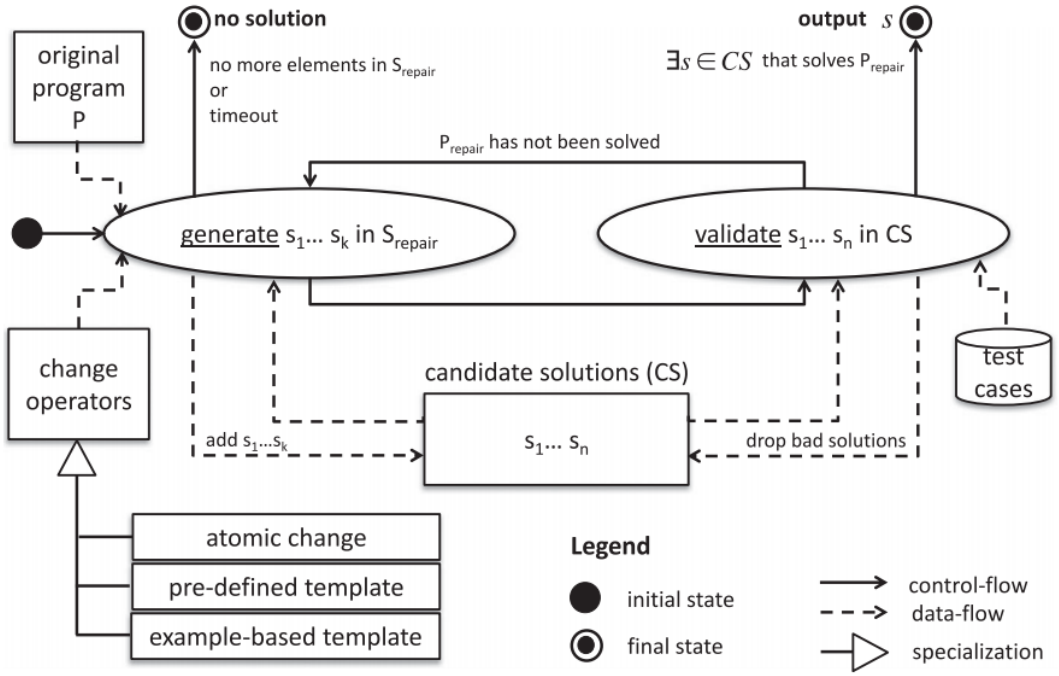

Figure 2.1: Generate-and-validate repair process from [18]

2.3.1.1 Atomic Change Operators

GenProg [53] is an example of a generic method that takes as input a defective C program and specifications in the form of test cases. It takes advantage of strategies of genetic programming to search for program variants that are not vulnerable to the initial defects of the program but keeps the desired functionality [26], the first variant to correct the defect and keep its original functionality is the *primary repair* [17]. The variants are represented by their AST, similarly to other approaches like Marriagent, RSRepair, and SCRepair [18], and a weighted program path and may be modified using crossover and mutations [53].

After the alternative program is created, the repair is minimized using delta debugging and tree-structured distance metrics, creating the *minimized repair*. The method has shown some promising results on some instances (examples from the 3 first papers), however, it is subject to weak test cases and to overfitting some test suites [48].

2.3.1.2 Pre-defined templates

Pre-defined template techniques take advantage of known snippets of code that can be used to repair faulty code. There are two main strategies adopted: search-based and brute-force techniques.

Search-based is usually used in situations where multiple non-trivial changes need to be made in multiple places in a systematic manner [18]. The search is appropriate since it must find and replace specific patterns that occur repeatedly throughout the code. Since it has such a specific use case, this approach has not been as explored as much as its counterpart and has been mostly associated with its implementation on concurrency faults. For example, ARC [24] to generate repair candidates for concurrency faults using pre-defined templates and genetic programming.

On the other hand, brute-force techniques use various pre-defined templates of various different types of fault and apply them to strategic locations until the program is fixed or times out. By having a diverse set of examples, brute-force approaches are not as bounded to specific fault models.

2.3.2 Semantics-Based Program Repair

Similarly to Generate and Validate, the repair process described in Figure 2.2 follows the same structure: it starts out by analyzing the program to understand what its behaviors should and should not be, it then uses the rules and constraints learned in the first step to generate potential fixes which will finally be confirmed to be valid or not.

Unlike Generate and Validate, semantic-based program repair focuses on fixing smaller and less generic bugs [41].

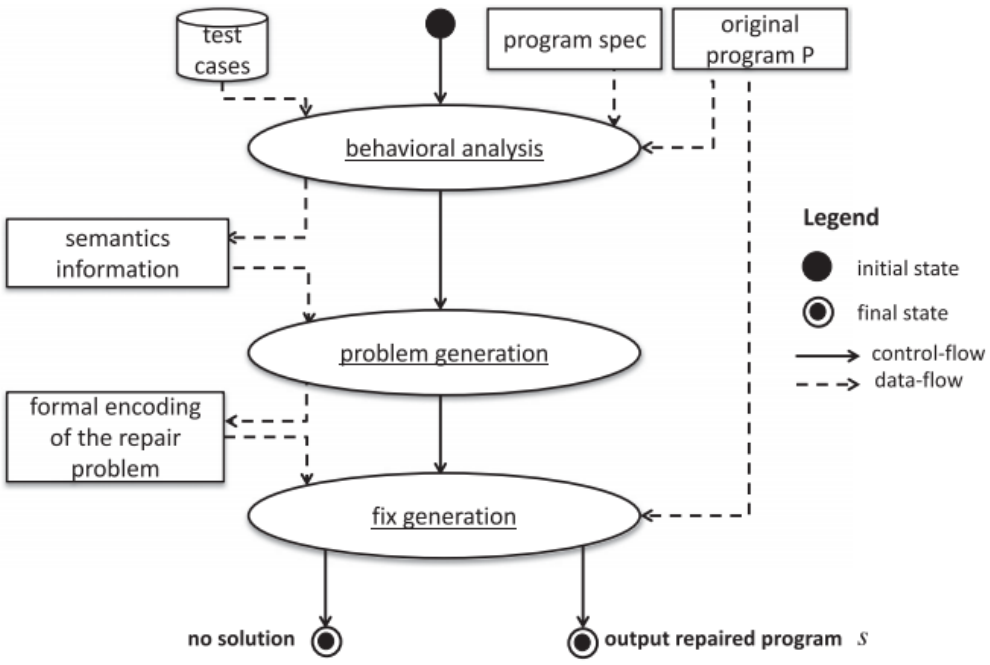

Figure 2.2: Semantics-driven repair process from [18]

2.4 Localization Strategies

To better decide where the mutations should be applied, this section describes tools and techniques used to find the locations in which the mutations are more likely to produce favorable outcomes.

As previously mentioned, in Automated Program Repair, fault localization techniques are used to find which code statements are the cause for the fault in the program. However, since our scope focuses on generic optimization, we will analyze the tools and techniques designed for code optimization. In software optimization, the most effective place to optimize is the process that is the least optimized and thus considered its bottleneck. In this section, we will explore the tools used to the location of these bottlenecks in programs.

Petsios *et al.* [39] show how worst-case scenario inputs can be found by using resource-usage-guided evolutionary search techniques by creating *SlowFuzz*. Their goal is to find which inputs trigger execution times with complexity much higher than the average case. To reach that goal, *SlowFuzz* tests various inputs for their execution path length and uses these results to guide the mutations for the next iteration. Furthermore, *SlowFuzz* is flexible enough to take into account other metrics such as memory consumption and even energy costs related to CPU usage.

Lemieux *et al.* [28] present *PerfFuzz*, designed and implemented a fuzz-based algorithm to find worst-case scenario input for programs. The algorithm uses the control-flow graph (CFG) of the input program and multiple initial inputs, called *seeds*, to find which sections of the program are the most expensive for each specific input. In each iteration, PerfFuzz decides whether a seed should be considered for mutational fuzzing. This decision is probability and based on how the input affects the performance (e.g. if the input results in new code coverage or if the number of edges visited increased). In comparison to the previously mentioned *SlowFuzz* which only produces one input from one parent input, *PerfFuzz* generates thousands of inputs for each parent and decide which input to prioritize using the concept of *favored inputs*, which are inputs that maximize the execution count *of at least one CFG edge*. Furthermore, the two algorithms were tested against the same data set and *PerfFuzz* outperformed *SlowFuzz* at finding hot spots, which was expected since its what it is tailored for, and also outperformed *SlowFuzz* in finding inputs that maximize the total path length, which is what *SlowFuzz* was made specifically for.

Poshyvanik *et al.* [20] present *FOREPOST*, a black-box software testing tool that extracts rules from executing traces of programs using machine learning. *FOREPOST* then uses these rules to guide the choosing of new inputs to find *intensive paths* with performance problems [31].

Shen *et al.* [47] introduce *GA-Prof*, an approach to find performance bottlenecks automatically by searching the input domain of the program. The main strategy used is to use a genetic program over the input of the program to find which parameters maximize the *fitness function* which, in this case, is the elapsed execution time of the application.

Toffola *et al.* [52] present *PerfSyn*, a generic framework that synthesizes a program to find a bottleneck in a given method. To find these bottlenecks, *PerfSyn* runs five sequential steps: first, it mutates the program, it then executes and gathers feedback, learns which of the mutations influenced the method's performance the most, and, finally, explores which new mutations can be applied to find a better solution.

All of the discussed tools leverage input given to a program in order to find a possible bottleneck in the program. Similarly, *SlowFuzz* and *PerfFuzz* use *fuzzing* techniques over the input and assess a mutation's quality based on the number of operations made from start to end. However, after running both algorithms over the same dataset, *PerfFuzz* outperformed *SlowFuzz* in both inputs that find higher hot spots and inputs that maximize the total path length. Like the previously mentioned algorithms, *FOREPOST*, *GA-Prof* and *PerfSyn* all search for the bottleneck by finding new inputs. However, and different from the fuzz-based approach taken by *SlowFuzz* and *PerfFuzz*, these algorithms take advantage of *rules extracted from executing traces*, *genetic programming* and *mutation operators*, respectively.

2.5 Live Programming

Live programming, used interchangeably alongside the term *real-time* throughout this work, generally depicts an environment in which there is a constant feedback cycle where the environment provides information to programmers about the program being developed [51, 4]. The goal of this concept is to steer away from the traditional development cycle in which a developer needs to compile and run the program to get feedback about the program's behavior in run time. This concept was initially applied in what was then regarded as an *Interactive Environment* for *Lisp*, which took advantage of nonstandard program structures and program development methods [44], Smalltalk [23] and visual languages [50, 51].

To further understand the core fundamentals of implementations of live development environments, this section will review recent implementations of live development environments to find the most important aspects of these environments, their challenges, and their differences in regards to more traditional development environments.

Lemma *et al.* [29] believe that existing IDEs are not ideal for live programming since they were not created with this concept in mind. Consequently, they propose to concurrently develop not only a programming language but also an environment that is designed with live programming in mind, named *Moon*. To reach their final goal of a live programming environment, they present three key ideas: (1) *Entities Visualization*, which has as a goal to portray entities of the program into *Representations*. These *Representations* represent the programs' entities in a more human-perceptible format and it is intended for basic representations to be integrated into the system while also giving the opportunity to users to create their new custom representations; (2) *State Visualization*, they argue that information about the whole state of the program is valuable to spot early errors.

In a prototype, the ecosystem takes advantage of real-time compilation to pinpoint the localization of the errors with the usage of syntax highlighting. They also suggest that the approach can be significantly improved by making this analysis at every change in the abstract syntax tree; (3) *Evolution Visualization*, which focuses on how, in a collaborative setting, important it is to understand the overall evolution of the system.

Kulbeka *et al.* [25] conducted a study on an analysis of 17 programming sessions in Pharo [9] conducted by 11 participants, composed by a Bachelor Student, a Master Student, Ph.D. Students, Professors, and Professionals.

When analyzing how the participants engaged with *Pharo*, a maturer live programming environment, *Kulbeka et al.* noted that some participants, despite using the liveness techniques, were not successful either because they *lost time* waiting for feedback which was not received, or by using an approach that ultimately would not be sufficient to solve the given task. On the other hand, there were also cases where developers used liveness techniques to check their assumptions while they were performing the tasks. There is even a report of a participant which used the aforementioned techniques to check their progress after *every small change* which would instantly give them feedback if the behavior was as expected or not. Interestingly, they also found that the most advanced approaches available were the most under-used whereas the most simpler tools, such as the *inspector*, a tool similar to what is found in *Google Chrome Development Tools*[1], and the *playground*, a tool to write code snippets to explore their results, which were used by virtually all the participants.

To sum up, *Kulbeka et al.* concluded that liveness was used very frequently during the programming sessions and that simpler approaches were favored in detriment to more advanced and thus more technically challenging approaches.

Oney *et al.* [35] present a programming language and environment ecosystem, called *InterState* that focuses on writing and reusing user interface code. *InterState* includes a live editor that gives immediate feedback to the user regarding the state of the program to help programmers understand the program's behavior in run-time. This ecosystem avoids the traditional difficulties of real-time feedback through event-callback code by including state machines and constraints as fundamental language constructs. To test the efficiency of this environment, the authors conducted a comparative laboratory study in which they exposed 20 *experienced programmers* to two different implementation tasks, one in *JavaScript* and the other in *InterState*.

Participants were more than twice as fast implementing the drag lock task in *InterState* comparing to the *JavaScript* alternative. For the second task, an image carousel, the participants once again finished the task in less than half the time with *InterState* comparing to the time it took to implement the image carousel in *JavaScript*. Users in the experiment described the *InterState's* visual notation as *intuitive* and *clean* and it is reported that nearly every user

[1] Google Chrome Development Tools: https://developer.chrome.com/docs/devtools/ Retrieved on 02/07/2021

credited their quickness of debugging throughout the task to *InterState's* ability to display real-time application state and live property values.

Fischer [16] introduces Circa , a language, and platform designed specifically to take advantage of having the state of a running program during the development process and use the information to improve the ability to create and manipulate code. This environment specifically focuses on dataflow-based programming model that works by representing the program as a directed graph of terms.

With this representation, each term has a set of inputs and a function that specifies how to calculate the output. Given this implementation, it is easy to backtrack a program result to find which functions and inputs affect the final outcome of the program and consequently, take advantage of visualization techniques to aid the user.

Riedl-Ehrenleitner *et al.* [42] define a novel approach to detect inconsistencies between design models and source code. *Model-Driven Engineering* [45] promotes the abstraction of concepts into model artifacts with the intent to simplify and express domain concepts efficiently. Moreover, and similarly to traditional compilers, model-to-code *transformations* [46] can be applied to these models to automatically generate source code, these transformations are however not perfect [56]. To solve this discrepancy, the authors propose *Model-and-Code Consistency Checking* (MCCC) to detect, in real-time, differences between consistency in the source code and the model which the source code derives from.

Figure 2.3 illustrates the MCCC framework overview. Once the source code, the conceptual models, and the consistency rules are defined, the *Integration Layer* can serve as an intermediate level that abstracts the concepts from both the *Model* and the *Code* and allow the *Consistency Checker* to apply the previously mentioned rules and check for discrepancies. This check can be applied with different strategies, e.g, prompted manually, before a developer decides to public changes, at predefined intervals, or even with incremental checking at the finest granularity by checking after every atomic change. Once the inconsistency is detected, a automatically derived repair may be suggested directly to the developer.

Oney *et al.* [36] presents *Euclase*, a visual live development environment focused on creating interactive web applications. This environment uses constraints combined with finite state machines to define how an object in *Euclase* behaves. Being a live development environment, as a developer changes the source code of the program, the changes are immediately reflected back to the developer since the program is always executing.

Regarding the importance of live environments, *Oney et al.*. found three reasons: (1) *Beginner friendliness*, they argue there is a relevant barrier to beginner developers who are more prone to make syntactic errors and often will lead to semantic errors. They further hypothesize that *by providing a live environment, we can help beginners better understand their programs* which is supported by their assumption that a beginner developer is likely to become discouraged after facing a lot of errors upon code compilation; (2) *Quick evaluation*

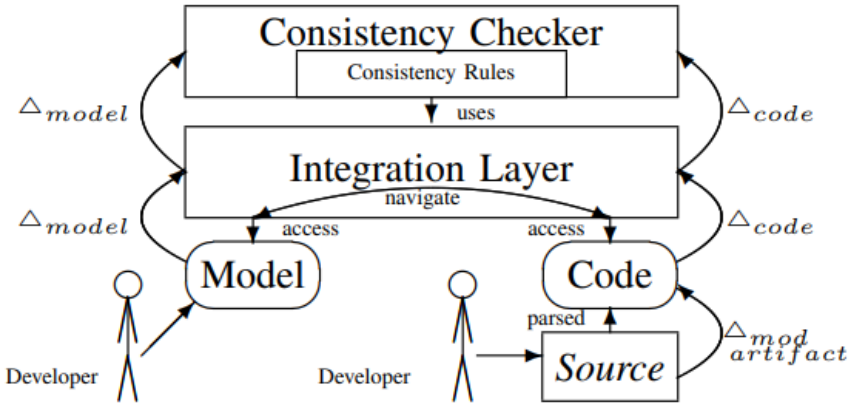

Figure 2.3: Model-and-Code Checking framework overview from [42].

describes how important it is for designers to quickly have feedback regarding the application's *feel*, meaning not only taking int account the visual aspect but also the interaction aspect, in contrast to its *look* which is, according to previous research [33], more effectively done with sketches or drawing applications; finally, (3) *Quick experimentation* claims that experimentation is crucial when designing an application. As previously mentioned, while present-day's development environments are great to draw and sketch the application's *look*, they do not offer good support regarding the application's feel, [21] and this lack of support is reflected in the lack of an environment which can be used to test the *feel* of the application in real-time. The authors couple this reason with a simple yet powerful example to describe this struggle: in a scenario in which a designer wants to tweak the *scrolling "friction"* to better suit the interaction in the given context, the designer can use *Euclase* to iteratively modify the *friction* parameter to see the results immediately in contrast to the traditional tools in which the program would have to be re-run.

Finally, the authors reflect on the design challenges of a live environment and they found that using a live development environment not always mean a quicker feedback loop between the developer and the application. To back up this statement, they give the example of when a user is working on a feature that requires the application to be loading, they do not have a better way to force the application to load in comparison to the standard edit-run loop.

Following the literature review regarding *Live Programming* resulting in the presented analysis, another work was suggested by experts in the studied field and is now analyzed.

Campos *et al.* [11] hypothesize that *using a live Automated Program Repair improves the speed and final results of code patches* as well as present *pAPRika*, an Automated Program Repair tool that is implemented as a *Visual Studio Code* and, automatically, proposes *semantic code suggestions* by leveraging existing unit tests as specifications. To guide their research, the authors focused on answering three research questions: (1) *Do developers fix an error faster*

when using a live APR tool?, (2) *Do developers write solutions that are significantly different from those generated by our approach?*, and finally (3) *Do developers understand why solutions generated by live APR tools work before accepting them?*. To test their hypothesis and answer their research questions, and using the tool implemented for this purpose, the authors conducted a preliminary empirical study with 16 participants in a crossover design.

The implementation of the extension relies on unit tests as specifications and uses a mutation-based approach, like the one used by *Debroy et al.* [13], to generate variants that are then tested against the specifications to assess if the mutation is valid or not. A patch, to be considered valid, has to pass all the test cases which make all the suggestions generated *complete patches*.

In the empirical study, the authors constructed a dataset of problems originally from the comments of the `r/dailyprogrammer` *subreddit* and created the complementary test suit. The problems in question were classified into one of the following three types: (1) *immediate* task where a bug is present in the code and a patch is immediately found by the tool; (2) *nonimmediate* task where a bug which is present but the tool can only detect it after a missing functionality is written (which is a direct consequence of the tool only suggesting *complete patches*; and finally, (3) *nonpresent* task where a bug is not present in the code, but the participant is required to write some code which might contain a bug that the tool may be able to fix. When analyzing the empirical study's results, the authors found that *users are faster* to reach a solution when using an Automated Program Repair tool. When analyzing whether the solutions from developers are any different from the automatically generated ones, the authors found that the solutions are *sometimes different* and have identified situations in which the solution generated by the APR tool is considered to be worse in terms of its *naturalness* [22]. Finally, regarding whether the participants understood the patches suggested by the APR tool, their findings were *counter-intuitive* since the results from a survey that was coupled with the tasks *directly contradicted* their own observations. To sum up, the authors consider having strong evidence supporting their main hypothesis due to the positive results regarding the speed of the participants.

Live development environments provide a powerful feedback loop between the developer and the environment. Environments such as *Moon* [29], *InterState* [35], *Circa* [16] and *Euclase* [36] heavily rely on visual representations of the program's state or other characteristics which may be edited in real-time to allow the user to get instant feedback on those changes. This feedback may include information such as if the program is running successfully or it can even be as subtle as getting a real-time *feel* of how the changes in the source code affect the how a user may interact with the program. On the other hand, environments like *Pharo* used in [25], the *MCCC* implementation [42], and *pAPRika* [11] rely more on code suggestions and text feedback, either by exposing additional information or facilitating an interactive interface that can be used to query the program's state and other characteristics.

Finally, despite the conventional definition of live programming being generally referred as the ability to edit an executing program, tools like *pAPRika* [11] and the seen *MCCC* implementation [42] skew away from this definition by implementing triggers that can be set off by the developers at any time to run an instance of the analysis engine and then give feedback. One downside of this approach is that if the process of generating feedback is too slow, the real-time part of live programming may be jeopardized.

2.6 Summary

Section 2.1 (p. 5) describes the methodology used for this literature review, including the databases and the procedure used.

Section 2.2 (p. 6) and Section 2.3 (p. 6) present a brief overview of some strategies used in Automatic Programming and Automated Program Repair which may be useful when implementing an Automatic Program Optimization tool. In Section 2.4 (p. 9), we analyze localization strategies that are used to locate the code statements which are more likely to be the bottleneck of the program.

Finally, in Section 2.5 (p. 10), we analyze tools and environments that implement some notion of live programming.

Chapter 3

Problem Statement

This chapter details our problem. In Section 3.1, we define our problem and we make a brief reference to how pAPRika, a real-time APR tool works. Secondly, in Section 3.2 (p. 18), we present the main hypothesis we aim to validate as well as the main research questions that guide this work. Finally, in Section 3.3 (p. 19), we outline the methodology we will use to validate our work.

3.1 Problem Definition

The majority of tools and techniques for code optimization are very specific to what they are trying to optimize. This specification often brings the best results possible, however, it makes the tool's ability to be extended a considerable limitation.

This dissertation aims to create an engine that explores the mutation space around an already functional software with the intention to find mutations that may improve the code.

In order to achieve this goal, we will extend a tool created by *Campos* in [11] because not only it is a live programming environment based on mutation operators, but it is also followed with an empirical study that shows evidence that the environment leads developers to faster development times. The *Visual Studio Code* extension developed in the previously mentioned work, pAPRika, uses mutation operators in semantically incorrect programs in the hopes to find a variant of the program that passes all the given sources of truth, in this case, unit tests.

As described in Figure 3.1, once the extension is activated and the tests fail, mutations are generated and tested to check if the tests started passing. Once a valid mutation is found, the mutation is suggested to the developer which decides if the patch should be accepted or not.

However, this extension was made purely for APR and is not prepared to include other metrics other than program correctness.

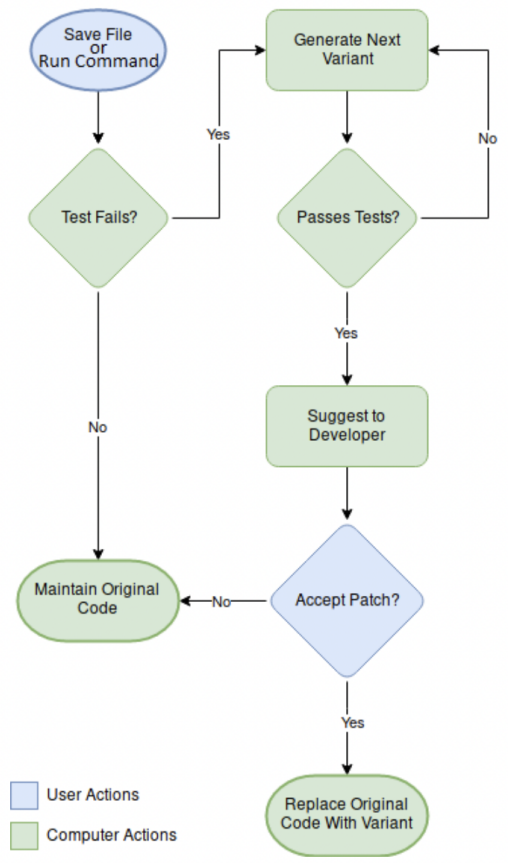

Figure 3.1: Flowchart of pAPRika from [10]

3.2 Main Hypothesis and Research Questions

To understand the impact of automated programming in a developer workflow, we intend to study how an Automated Program Optimization tool can impact the speed of both correcting programs with bugs and optimizing them in order to mitigate the significant maintenance costs. Consequently, The goal of the work of this dissertation is to validate the following hypothesis:

> *"Using a real-time Automatic Program Optimization tool improves the speed and final result of code solutions."*

A mutation-based real-time Automated Program Optimization tool should be able to find mutations that are semantically equivalent to the original code and be able to decide which changes it should apply and which it should not base on the priority that the user gives to the input metrics while still being able to behave as an Automated Program Repair tool.

To support the presented hypothesis, the following research questions were identified to guide the research:

RQ1: *Are users faster in reaching a patch when using a real-time Automatic Program Optimization tool?*

To understand if the usage of an Automatic Program Optimization tool really contributes towards the cut in development and maintenance costs we need to first understand if its usage has a significant impact in a developer's workflow. Consequently, we aim to find if the usage of an APO tool makes the developers faster and thus more efficient leading to lower development and maintenance costs. We aim to study this by challenging developers with similar technical knowledge and experience with tasks and study the difference in times between developers using an APO tool with times of developers without such a tool.

RQ2: *Are users aware of the rationale suggestions generated by an Automatic Program Optimization tool before accepting them?*

Despite the speed of development being a very important metric when trying to optimize the process of software development and maintenance, the health and sustainability of the project is a necessity for long-term projects. To understand how the usage of an APO tool may affect the sustainability of a project, we want to find out if a developer using an APO tool understands the patches proposed before accepting them. By understanding the patch we assume that a developer is not inserting in the codebase patches that are hard to understand for future developers and maintainers and thus not as sustainable in long term.

RQ3: *Are solutions programmed by human developers different from the solutions generated by an Automatic Program Optimization tool?*

To understand the quality of the suggestions made by an APO tool, we plan to study the differences between patches generated by developers without the tool and patches generated by an APO tool. With this in mind, we intend to expose developers with similar technical knowledge and experience to some coding challenges and study the difference between solutions that developers create and the solutions generated by an APO tool.

3.3 Validation

To evaluate the proposed hypothesis, we plan to conduct an empirical experiment with around 20 participants. These participants are to be divided into two groups to understand the differences between how developers perform when using an Automated Programming Optimization tool and when not using an APO tool.

With this experiment, we plan to study the differences in times to reach a valid patch to a proposed task between developers using an APO tool and those not using it, considering a valid patch a change to the code that either turns a semantically incorrect program to a correct one

(program repair) or attempts to optimize the program (for other metrics like energy consumption, performance, memory usage, etc...).

Moreover, we want to understand if the developers are aware of the meaning of the suggestions provided by the APO tool. With that in mind, we compare the answers regarding the understanding of patches from the surveys with the final code solutions submitted by the participants. If both the survey answers show positive feedback from the developers and the final code solutions suggest that the patches are read and understood before being accepted, we can claim that there is significant evidence that the users understand the feedback from the APO tool.

Finally, to study if the human developers produce different solutions from the patches generated by an APO tool, we use the final code submissions from the participants and analyze the differences between solutions generated by developers using the tool with solutions when the developers have no access to the tool. By making this comparison, and taking in mind the list of proposed solutions by the APO tool, we can study how these suggestions impact the decision process of the developer regarding the final code solution.

3.4 Summary

In Section 3.1 (p. 17), we detail the problem definition and frame our work in regards to the work of *Campos Campos19*. In Section 3.2 (p. 18), we propose our hypothesis and the research questions, as well as a brief explanation for each question, that guide this work. Finally, in Section 3.3 (p. 19), we explain how we intend to validate our work and justify why our methods may be used to answer our research questions, and consequently assess our hypothesis.

Chapter 4

Framework Implementation

This chapter describes the details of the implementation, it contains both an overview of the *Architecture* design in Subsection 4.1.1 as well as their component details in Subsection 4.1.2, Subsection 4.1.3 and Subsection 4.1.4. Furthermore, it presents a description of how the user interacts with the application in Section 4.2 (p. 29).

4.1 Architecture

Despite our research in *Localization Strategies*, the scope of this work focuses on challenges that are simple enough that localization improvements are not significant and thus are not considered. The implementation of the framework is mainly divided into three distinct components: operators, environments, and strategies.

4.1.1 Overview

In order to have multiple functions and files being mutated at the same time, instances of mutations must be able to be tested simultaneously. Therefore, a *Mutation Instance* is defined as a data object with the necessary information to process a mutation. The life cycle of these objects in optimization strategies, where the program is already semantically correct, is documented in the sequence diagram found in Figure 4.1.

Once the user triggers the extension to search for mutations, the original code is executed and it is confirmed that it passes all the tests, the process stops if they do not. Afterward, and according to which operators were assigned to the current strategy, mutations of the original code are created and hence the start of the *Mutation Instances*. Asynchronously, mutations are generated

by the *Strategy* and sent back to the server to make sure the new mutations still pass the tests, the mutations which do not are automatically discarded. The mutations are then sent back to the *Strategy* as a valid mutant (i.e. is semantically equivalent), now the strategy must measure if the mutation is good enough to be suggested. With that in mind, the mutation is sent to a previously defined environment which will run the mutation 10 times in conditions that enable a metric to be evaluated. Once the score is recorded, the mutation, and the newly recorded score, is sent back to the strategy which now has the to make an informed decision on accepting or not the suggestion. In this work, this decision is made by the means of a *t-test* where it is evaluated if the new score is statistically better than the previous best mutation found for this function.

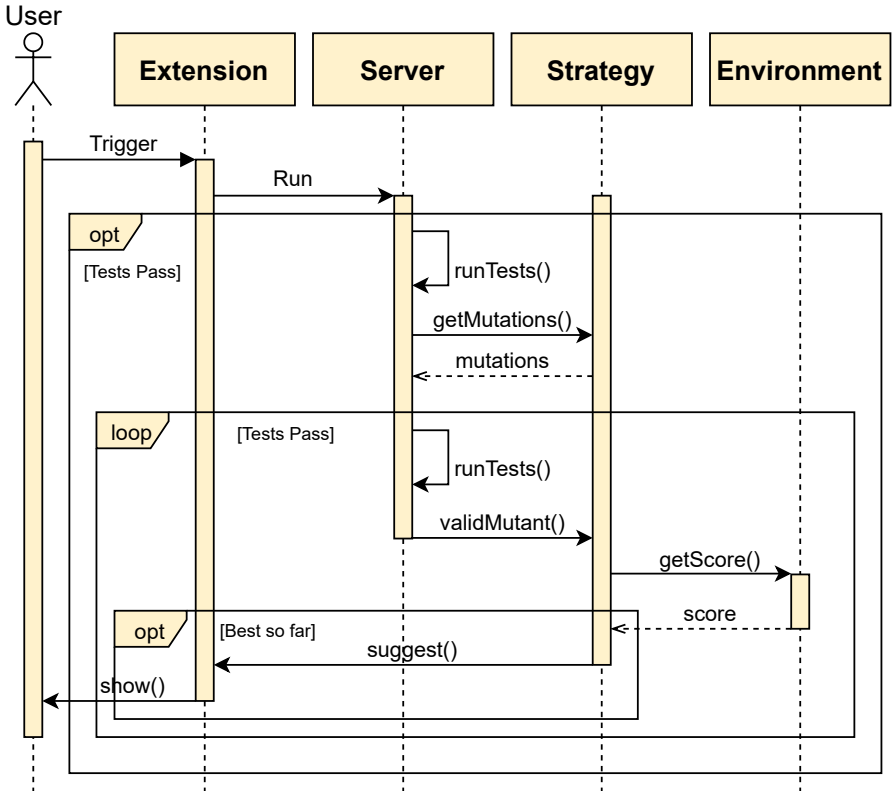

Figure 4.1: Sequence diagram of how a mutation is created and considered for optimization with the score being decided by a predefined run-time environment.

Optimizing the program correctness has a very similar life cycle with the exceptions of some smart optimizations that can be made. Contrarily to the previous optimizations, for a program repair to be useful, the original program must not be semantically correct, therefore the program is initially tested to make sure it fails in at least one test. The rest of the process is equal to the previously mentioned process with the exception that in program repair the scale for the score is binary, either the program is repaired or not. Therefore, once *validMutant()* is called, there is no need to check its score, because it is already known to be maximal. Consequently, the mutations can be directly suggested to the end-user without further need of a specific run-time environment.

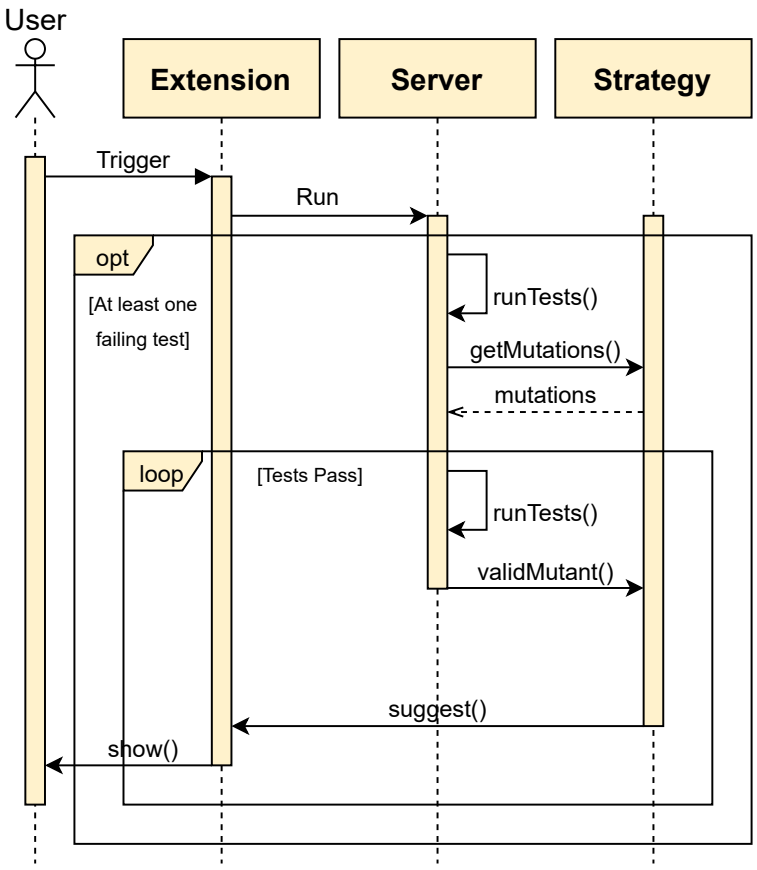

Figure 4.2: Sequence diagram of how a mutation is created and considered for program repair.

4.1.2 Mutation Operators

An operator is a mutation operator rule that defines how to modify certain features of the artifact undergoing mutations [14]. The goal of these operators is to create versions of the original program that only differ by one small change [49].

In this tool, the program follows the same strategy as *pAPRika* (*cf.* Section 2.3, p. 6), the code is traversed over through its Abstract Syntax Tree (AST) representation, and the mutations are applied in the nodes.

The traversal follows the TypeScript Compiler API [1] which allows to programmatically traverse the TypeScript Nodes and apply mutations. For each node, and to preamble the following implemented operators, we have access to its *left-hand side* and *right-hand side* children, *node.LHS* and *node.RHS* respectively, as well as other information about the node itself such as which *keyword* it represents.

In this work, we use the concept of *Replacement* as an object with the information needed to apply a mutation in the source code. This object contains information about the *start* and *end* position of the code in the original code that will be replaced once the mutation is applied.

[1] https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API Retrieved on: 29/06/2021

Moreover, the *Replacement* keeps information about the *oldText* and *newText* which is the string comprised from *start* and *end* and the newly generated mutation that will replace the *oldText*, respectively. Finally, the *Replacement* has a *code* as an identifier for implementation details (*e.g.,* the name of the file of the variation will depend on the mutation's identifier).

The goal with this work is not to get the best performance with the best mutation operators, the goal is to study how suggestions made an APO tool improve one's development workflow. Consequently, our mutations were loosely based on concepts from *Bentley's* work [7]. In his work, *Bentley* provides a pragmatic treatment of program efficiency divided into 6 major sections: *Space-for-Time Rules*, *Time-for-Space Rules*, *Loop Rules*, *Logic Rules*, *Procedure Design Rules* and, *Expression Rules*. However, some of these rules are not suitable for mutation-based operations. For example, *Loop Rules* describe strategies such as *Loop Unrolling* which rely on the fact that a large cost of short loops are caused by modifying the loop indexes and a solution for that would be to unroll smaller loops. Nonetheless, creating a mutation to translate this process would be a significant technical challenge and thus not suitable for this work.

In the following subsections, we present which mutations were implemented and our thought process behind the decisions made.

4.1.2.1 Contraction of constant variables

Based on *Exploit Algebraic Identities*, a strategy from *Expression Rules*, which consists of replacing a costly expression with one that is algebraically equivalent but cheaper, we developed a mutation that contracts constants.

The contraction of literal constants, exhibited in Algorithm 1, are made by recursively finding instances where both child nodes are able to be simplified (line 13), and then simplify said node. This operator only simplifies literal values and property access expressions (i.e. constants that require access to resolve).

The contraction of literal constants, as exhibited in Algorithm 1, is made to reduce the number of operations that the program must compute in order to reach a constant value in run-time. As described in the *pseudo-code*, a mutation is generated once the function associated with the mutation, `SimplifyMathConstants()`, is called. This function takes the current node information and the list with the replacements generated so far in the search. Right away, an auxiliary function named `GetConstantValue()` is called with the information regarding the current node. The node will be analyzed and there will be an attempt to simplify the children nodes in a recursive fashion:

First of all, if the node is already a numeric literal and thus a constant, the node is returned right away as seen in line 8. Secondly, it is checked if the node is a math operator (line 10) which boils down to checking if the node is one of the following operators: `/` for division, `*` for multiplication, `-` for subtraction, `+` for addition and `%` for the remainder. If the node coincides with one of the considered math operators, two recursive calls, one on each child node, are called to query the constant values of the node's children. If both these values are numeric literals (line 13), the new constant value is calculated and a new node created and returned, otherwise, the `NULL` is returned

to inform the node's parent that this node cannot be further simplified. Finally, in line 18, it is known that the node is not a numeric literal nor its children can be simplified into numeric literals and thus the function returns `NULL` for this node.

Algorithm 1 Contract simple Mathematical constants

```
1:  procedure SIMPLIFYMATHCONSTANTS(node, replacementList)
2:      newNode ← getConstantValue(node)
3:      if newNode then
4:          replacement ← Replacement(node, newNode)
5:          replacementList.push(replacement)                ▷ Save mutation
6:      return replacementList
7:  procedure GETCONSTANTVALUE(node)
8:      if isNumericLiteral(node) then
9:          return node
10:     else if isMathOperator(node) then
11:         leftNode ← getConstantValue(node.LHS)
12:         rightNode ← getConstantValue(node.RHS)
13:         if isNumericLiteral(leftNode) and isNumericLiteral(rightNode) then
14:             newValue ← calculate(leftNode, node.op, rightNode)
15:             return Node(newValue)
16:         else
17:             return NULL
18:     else                                    ▷ Not literal nor Math Operation
19:         return NULL
```

4.1.2.2 Keyword Mutation

Despite *Bentley's Rules* focusing on language-independent optimizations, there are also some language-dependent operators which take advantage of the different properties of syntax with similar behaviors. Loosely based on the concepts from *Space-for-Time Rules* which argue for additional information or by changing the information within a data structure, we decided to explore how different declaration keywords in *JavaScript* behave differently.

This operator explores the differences between the keywords *const*, *let*, and *var*. The main difference between these keywords is that *const* declares read-only constant, *var* declares a variable, which might be initialized and, *let* is similar to *var* with the exception that its scope is limited to the block where it is defined [1]. Consequently, when we have loops that are traversed a large amount of times with a *let* or *const* variable inside, the variable is created and destroyed each time the loop is traversed. On the other hand, since a *var* variable is global, it is not created and destroyed each time, it is only updated, which is cheaper than searching for all the variables in the scope for simpler programs. It is important to notice that these are not necessarily equivalent mutations since it depends on the non-existence of variables with the same name in wider scopes.

The mutation itself is detailed in Algorithm 2. Whenever a variable declaration keyword is found, the operator creates two mutations by replacing the current keyword with the other two possible keywords from the previously mentioned pool.

More specifically, whenever a *variable declaration* node is found and this mutation is activated, the function `SwitchVariableDeclarationKeyword()` is called with both the *node* information and the current list of the replacements found so far. Out of the possible declaration keywords that the node may have: `const`, `let`, and `var`, the algorithm will create two new replacement mutations for each of the two declaration keywords that *are not* the node's keyword. A new node is then created by copying the original node and replacing the keyword used as described in the function `VariableDeclarationExp()`. Lastly, the newly generated replacement is saved in *replacementList* and returned back to be used in the future.

Algorithm 2 Mutate the Declaration Keyword

```
1:  declarationKeywords ← [const, let, var]
2:  procedure SwitchVariableDeclarationKeyword(node, replacementList)
3:      for keyword ∈ declarationKeywords do
4:          if keyword ≠ node.keyword then
5:              newKeywordNode ← VariableDeclarationExp(node, keyword)
6:              replacement ← Replacement(node, newKeywordNode)
7:              replacementList.push(replacement)          ▷ Save mutation
8:      return replacementList
9:  procedure VariableDeclarationExp(node, keyword)
10:     newNode ← node.copy()
11:     newNode.keyword ← keyword
12:     return newNode
```

4.1.3 Environments

An environment is where the quality of the program, according to a given metric, will be evaluated. Once a *Strategy* sends a valid mutation to an environment, the code is either statically analyzed, or ran multiple times to get a sense of its behavior in run-time.

Our concept of Environment is specialized for situations where, for each consecutive running instance, there are three function calls: (1) one before the test is run, (2) then after the test is run, and finally, (3) a function that is able to interpret the collected data and calculate a meaningful score from it. However, to accommodate environments that either need to interpret run-time data differently or that require static analysis, the main Environment class can be extended independently.

For this work, two environments were implemented, one to measure the performance of a program evaluated in run-time and, likewise, an environment to evaluate the power consumption from running the tests. Regarding the **performance environment**, we use the *Performance API* [2] which

[2]Performance API: https://developer.mozilla.org/en-US/docs/Web/API/Performance Retrieved on 02/07/2021

provides information about timing-related performance. Specifically, and considering the previously mentioned functions defined for environments: (1) before running the tests the time elapsed since a reference instant, saved as *t0*, is recorded using `performance.now()`. Likewise, (2) the time is recorded once more after the test is run, saved as *t1*. Finally, (3) the score for a single run of the test is calculated by subtracting the two reference times previously recorded, *t1-t0*. Concerning the **power consumption environment**, the environment takes advantage of the powercap interface provided by the *Running Average Power Limit (RAPL)* [12], an interface that uses software power models to estimate energy usage based on hardware performance counters and I/O models. Specifically, in machines with processors that implement this interface, a file named `energy_uj` can be found in `/sys/class/powercap/intel-rapl:0` where `intel-rapl:0` is the name of the device. Similar to the *performance environment*, an initial value for energy consumption is recorded before the test is run, and a final one is recorded after the test finishes. Lastly, these values are subtracted from each other to calculate the energy consumption during the execution of the test in micro joules. It is worth mentioning that some activities using this interface advise caution regarding the overflow of this measurement in samples taken more than 60 seconds apart [2]. Due to the simple nature of our problems, the tests run much faster, and consequently, this threat was not considered however, in future work with more lengthy tests this should be taken into account.

Finally, since the time elapsed and the energy consumption is not measurements only regarding the tests in our extension but rather are metrics of the performance of the whole machine, for these values to be consistent, the variation of the number of computationally demanding processes running simultaneously should be kept to a minimum to assure the quality of the measurements.

4.1.4 Strategies

A Strategy defines how the server will apply mutation operators over the program and, after getting valid mutations of the original program, will decide if each mutation is good enough to be suggested to the end-user or not.

To implement a *Strategy*, one must implement a set of defined functions and callbacks defined by an interface to assure it behaves according to the system specifications:

- **getMutations()** The *Strategy* must define which of the operators available in the *Operators* module it should apply in the program to create a set of *Replacements* which will, later on, be used to create variants to be evaluated. It is worth mentioning that by adding more operators, although the *Strategy* becomes more powerful and more likely to find wanted variants, it also makes the search universe bigger;

  By implementing a new *Strategy* one must aware of these advantages and disadvantages and choose the operators according to the goals of the *Strategy*. For example, in this work, there is a very clear difference between the operators used in optimizing energy and performance and those used for program repair. Given that the goal of program repair is to create a new version where the program passes the tests, its semantic meaning must be changed.

However, in the case of the strategies that optimize energy and performance, the goal is to create equivalent variations that have better quality metrics;

- **validMutant()** When the *Strategy* gets a valid program variant, meaning a mutation that passes all tests, it must decide whether it is worth being suggested to the end-user or not. In this function, it is expected for the *Strategy* to ask an *Environment* how well the new variation performs and then comparing the results with other variants' performance before making any suggestion or to directly suggest the mutation to the end-user;

- **evaluateScore()** Working as a callback to the $Environment's\ getScore()$, in $evaluateScore()$ the *Strategy* has the chance to compare the score of a new variant with any information it might have stored previously about the performance of variations that have already been generated.

In this work, we use a one-tail *t-test* over the set of scores returned by the *Environments* to assure that there is a statistical difference between the two variants and that one variation does not just perform better by chance. After running the *t-test*, if the *p-value* is lower than *0.05*, we can reject the null hypothesis that both results are derived from two programs with similar performance.

4.1.5 Technical Challenges

As the number of mutation operators added to the extension increases, so does the number of variations and consequently, the time spent evaluating said mutations. Due to the simplicity of the empirical tests designed for this work, the maximum number of mutations per task was reasonable for the computing power of the used machine. However, for bigger programs with more nodes to be mutated, the number of variation files created and tested can easily prove to be too demanding for an average machine, making the use of the tool for the average developer unreasonable. Moreover, once the extension crashes from the overload of work, the residual files may affect future uses of the tool. To tackle these potential downfalls, we advise future researchers, who may want to create or extend a mutation-based framework, to create a management component to make sure the extension is as stable as possible.

Considering the aforementioned two factors, we advise for the creation of a **mutation pool**, which asynchronously takes information about which mutations to apply and is responsible to run them. Based on the design pattern *Object Pool*, this component could take advantage of the slow process of creating, editing, closing, and deleting files and could simplify the whole process by keeping a pool file descriptors that are used to test variations. With this method, we can do without the creation and deletion of a new file for each variant and we can easily assure that the maximum number of mutations being generated at the same time is manageable according to the systems' capabilities. This change would allow for quicker mutation testing and consequently the increase of the number of mutations being tested as well as the additional stability from the management of

resources. Furthermore, this component, by defining specific file descriptors in every run, would guarantee that past crashes would not have considerable impacts in future uses of the extension.

4.2 Visual Studio Code Extension

Our tool is extended from the work of *Campos* [11] and thus already has a way to suggest a mutation to the user and a simple way for them to accept the suggestion. In Figure 4.3 and Figure 4.4, we can see how a user, after hovering over an underlined section of the code, is able to read what the extension is suggesting and also accept the suggestion by clicking on *Quick Fix...* which automatically approves the patches and updates the code.

In Figure 4.3, we can see a suggestion of a constant variable contraction, as described in Subsection 4.1.2.1, where the expression `9.8 / 2` can be contracted to `4.9`. The Figure 4.4 illustrates the extension displaying multiple suggestions generated with the *Switch Variable Declaration Keyword* mutation, found in Subsection 4.1.2.2 where it suggests that keywords `const` and `var` have better properties than the current keyword, `let`, regarding the active *Strategy*.

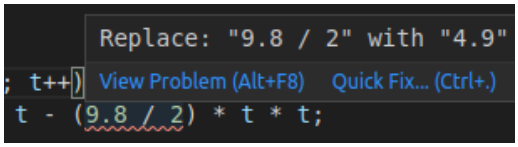

Figure 4.3: Code suggestion regarding a constant variable contraction.

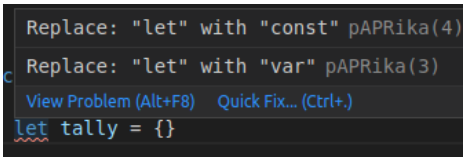

Figure 4.4: Code suggestion regarding a declaration variable mutation.

To accommodate the new changes which allow users to change between strategies, a new setting called *strategy* has been implemented. This option can be changed by browsing the extension's settings and using the toggle to change between *Program Repair*, *Energy Optimizer* and *Performance Optimizer* as seen in Figure 4.5. Once the strategy is set, the user only has to save the file by pressing *CTRL+S* for the extension to run and look for suggestions.

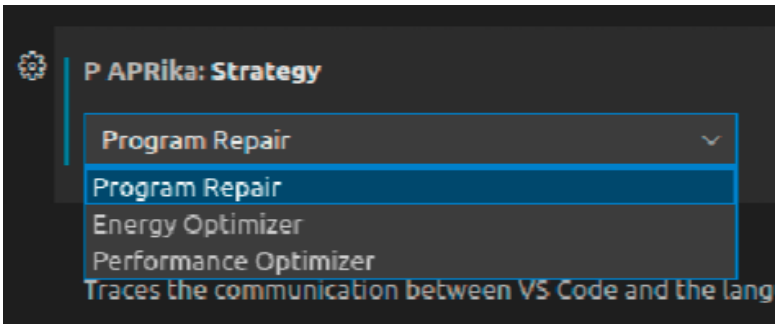

Figure 4.5: Extension's drop-down setting to select the strategy

4.3 Summary

This chapter describes the proposed solution and the details of its implementation including the framework architecture overview and the *VSCode* extension. In Section 4.1 (p. 21), we explain the general architecture of the framework by explaining its components, the *Mutation Operators*, the *Strategy* and the *Environments* and how they interact in order to reach the end goal of making a valid suggestion to the end-user.

Finally, in Section 4.2 (p. 29), it is described the functionalities already implemented in *Campos'* work [11], and the features implemented to integrate our *Automated Program Optimization* through *Visual Studio Code*.

Chapter 5

Empirical Evaluation

This chapter focuses on presenting the empirical evaluation of the tool developed in this work. The methodology used in this study is presented in Section 5.1. Secondly, the results of the experiments are presented in Section 5.2 (p. 36) where the results are exposed, analyzed, and discussed. Finally, the threats to validity are presented in Section 5.4 (p. 48) and the section is summarized in Section 5.5 (p. 49).

5.1 Methodology

The goal of the study is to evaluate the performance of developers during their workflow when using our *Visual Studio Code* extension and to compare it to the performance of a similar workload without the use of the tool. This evaluation is made by measuring the time taken to solve each problem, recording the usage or not of our tool, and assessing the final code for each task.

As illustrated in Figure 5.1, participants start off with the **Setup** of the experiment. In this stage, the participant is given a brief overview of the structure of the experiment with details regarding how the practical experiment is divided into two parts and that there will be is of surveys interspersed with the practical activities. The instructions are simple: one only has to follow the steps in the surveys. The survey will eventually redirect the participant towards the tasks, once the tasks are finished, return to the survey to continue. Still, during setup, it is explained to the users what our Automated Program Optimization tool does, and how to use it. Furthermore, it is indicated in which half of the practical experiment the participant has access to the extension and which one they do not. Finally, the participant is instructed to install *RustDesk*, the remote desktop

software used in this experiment, and to connect to the test machine. Afterward, the user is given access to *Background* survey and the experiment officially starts.

During the **Background Survey**, the users are asked about their technical knowledge and levels of comfort regarding the techniques and technologies used during the experiment (*cf.* Subsection 5.2.1, p. 36). Afterward, the participant is instructed to go to *RustDesk* and solve the tasks from the **Practical Experiment: Problem Set X**. Once the tasks are solved and the participant goes back to the survey to answer to **Practical Survey: Problem Set X** which is a section regarding the problem set just solved. At the end of this section, the user is redirected back to the practical environment to solve the tasks from *Problem Set Y*. Likewise, the practical experiment of *Problem Set Y* is followed with a survey regarding the undertaken experiment. A brief overview of the tasks can be found in Subsection 5.1.2 and the tasks can be found in Appendix A (p. 55). Finally, the user is asked to answer the **Post-Test Survey** which collects the overall feedback from the experiment (*cf.* Subsection 5.2.3, p. 45). Once the *Post-Test Survey* is answered, the experiment is over, it is assured that the surveys were answered and submitted, the final solutions are archived and lastly, after some words of appreciation, the remote software connection can be terminated.

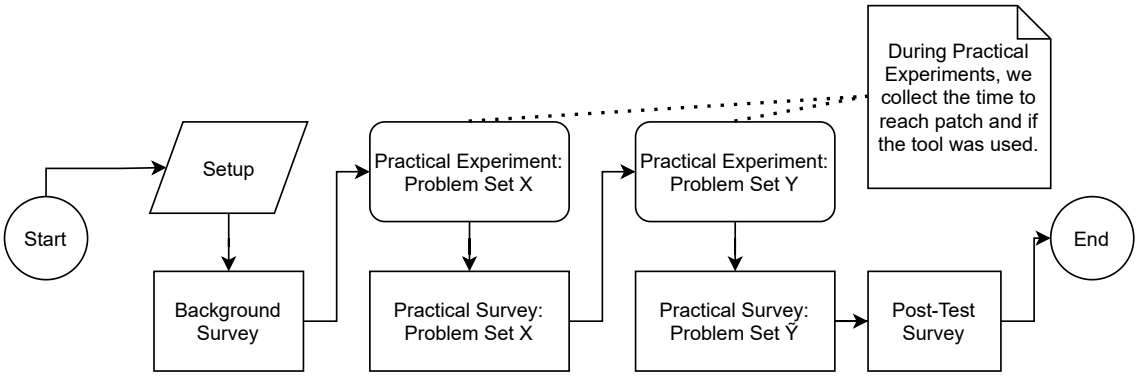

Figure 5.1: Diagram of the empirical experiment methodology conducted in this work. During the practical experiments, the raw data regarding *time to reach patch* and the *tool usage*, found in Table B.1 and B.2, was collected.

5.1.1 Plan

Participants — All the participants must be enrolled in university in a computer science-related field of study at the time of the test. Furthermore, it is required that all the participants had some previous experience with *JavaScript* to reduce the impact of having to learn new language-specific syntax during the experiment;

Duration — The estimated duration for the experiment, including both taking the survey and finishing the tasks, is 30 to 40 minutes. However, each task has a *timeout* set at 7 minutes to prevent the experiment from taking too long and compromising the users' participation quality;

Environment — To lessen the potential danger the volunteers would be exposed to by making these experiments live in person during a global pandemic, this empirical test is to be performed online. Consequently, by having a mandatory standardized development environment, the setup requirements substantially increase and therefore decrease the number of volunteers that are able to perform the experiment. Furthermore, one of the strategies of the tool requires a specific Intel Linux Driver that provides live information about the hardware energy consumption, which is not a standardized interface among the hardware manufacturers and is likely to reduce the volunteer pool even further.

Finally, and to reduce the resistance one might have to install Remote Desktop Software, we use *RustDesk's* [1] solution because it is multi-platform and open-source. These decisions are made to minimize the variables in the experiment, which are further explored in Section 5.4 (p. 48).

Every volunteer uses the same machine running Ubuntu 20.04.2 LTS with an Intel (R) Core (TM) i7-8700 CPU @ 3.20GHz, *Visual Studio Code* version 1.57.1, the tool resulting from this work and has access and permission to browse the Internet;

Procedure — To compare how developers perform with the tool and without it, we divide the 14 participants into two groups of 7 members. Since there are no guarantees that all developers have the same degree of technical knowledge, the two groups have both to be tested using the tool and not using the tool to have comparable data. Moreover, to assure that the previous usage of the tool does not affect future behavior, the two groups use the extension in different stages of the experiment. While one group uses the tool after being exposed to a problem set without using the tool, the other group starts the experiment using the tool and finishes it by solving a problem set without using the tool. To accommodate all these requirements, we divide the test into two parts per the following structure:

First Part:

- *Group A* attempts to solve the tasks from *Problem Set X* **without** the *VSCode* tool.
- *Group B* attempts to solve the tasks from *Problem Set X* **with** the *VSCode* tool.

Second Part:

- *Group A* attempts to solve the tasks from *Problem Set Y* **with** the *VSCode* tool.
- *Group B* attempts to solve the tasks from *Problem Set Y* **without** the *VSCode* tool.

Collected Data — During the empirical evaluation, three things are to be recorded for each task: (1) the time it takes developers from when they are introduced to the new task until they reach either a solution or an initial optimization, and, for the tasks the users are allowed to use our tool, (2) if they use any suggestion made by the tool. Finally, after the evaluation is over, (3) all the solutions for the tasks are saved;

[1] RustDesk — An open-source remote desktop software, https://rustdesk.com/

Survey — To have accurate and continuous insight from the user during the evaluation, a survey is carried out alongside the tasks. The survey is divided into four main sections:

1. **Background** Focuses on gathering information about the expected technical knowledge of the user as well as their experience using the tools required in the experiment.
2. **Practical: Part1** To be answered directly after finishing the tasks from Part 1, this section is meant to understand the users' perspective of the problems and their performance in the test.
3. **Practical: Part2** Similarly to *Practical: Part1*, this section is meant to better understand the users' perspective on the tasks but from the second half.
4. **Post-test** After finishing all the tasks from *Practical: Part 2*, the user is asked about their overall satisfaction and experience when using the extension, as well as their opinion on what is more important in the tool and what they would like improved.

Depending on which group the user is assigned to (A or B), the *Practical* sections will be slightly different depending on the usage or not of the tool to get further insight on its usability and performance. To measure the participants feedback to each statement, they are required to react to each statement in a 5 point *Likert scale* [30] comprised by: *Strongly Disagree*, *Disagree*, *Neutral*, *Agree* and *Strongly Agree*.

5.1.2 Tasks

We created a dataset specifically for this study to evaluate the tool's impact on a developer's workflow. There are plenty of benchmarks for both program repair [2] and program optimization [3] however, these were created to access the quality of the solutions designed whereas the goal of this work is to study the utility of real-time suggestions during the development workflow. Consequently, the tasks in the experiment are either a simplification of the optimization problems [15] (Tasks 3) or adaptations from simple program repair tasks already curated by *Campos* in [11] (Tasks 1, 2, and 4).

Task 1

The first task is meant to introduce the user to the extension. The user is asked to implement a version of either *slice* or *substring* but with slightly different interval restrictions.

In **Problem Set X**, the user is asked to implement a version of *JavaScript's substring* where the second index should be included.

In **Problem Set Y**, the user should implement a version of *slice* where the second index should be included in the final array.

[2] http://program-repair.org/benchmarks.html Retrieved on: 29/06/2021
[3] https://www.mcs.anl.gov/~more/cops/ Retrieved on: 29/06/2021

Task 2

The second task presents to the user an already fully working program. Again, the goal of the volunteer is to optimize the performance of the given function. For these tasks, it is expected for the users to use a *Declaration Keyword Mutation*.

In **Problem Set X**, we introduce a fully working Bubble Sort implementation where every variable is defined with the *let* keyword. The user is expected to either define *len* or *tmp* as *const* or realize that the usage of *var*, despite not as safe because of its global scope, prevents the variable from being defined on every single loop interaction and result in faster mutations for the variables *i* and *j*.

In **Problem Set Y**, the problem is adapted from [11] where the function calculates a set of scores based on a string originally from the */r/dailyprogrammer subreddit*[4]. Analogous to its counterpart from *Problem Set Y*, the goal is to induce the user to take advantage of the different properties of the *Declaration Keywords*.

Task 3

The goal for the third task is to evaluate how comfortable the user is to optimize a given function regarding its energy cost.

Pinto *et al*. [40] identified eight general strategies to reduce energy, but most of the strategies are too complex to be improved through the use of simple mutation operators and often use hardware-only concepts, which are out of scope for this work. Therefore, considering the limitations of mutation operators and assuming the energy consumption of our program is limited to the sum of the costs of its operations, the users are introduced to programs with a lot of operations, with the goal of simplifying the instructions and thus saving energy.

In **Problem Set X**, the user is asked to optimize a function that calculates the positions of an object in a specified interval. By rearranging the expression and/or simplifying some constants, reducing the number of operations necessary to execute the program is possible.

In **Problem Set Y**, the goal is to optimize a function that calculates the area of the *Gabriel's Horn*, a geometric figure with an infinite surface area but finite volume [54]. Following the same rationale as in the previous *Problem Set*, the instructions can be rearranged, and some operations can be simplified.

Task 4

For the fourth and final task, the users are asked to finish implementing a function. However, following the instructions to finish the implementation is insufficient as there is an underlying bug in the program. Both tasks are adapted from *Campos'* work [11].

[4] https://www.reddit.com/r/dailyprogrammer/comments/8jcffg/20180514_challenge_361_easy_tally_program/ Retrieved on 29/06/2021

In **Problem Set X**, the problem is adapted with the goal of doing the reverse of a factorial, meaning that if $n! = m$, then n is the reverse factorial of m. This problem can be found in the */r/dailyprogrammer subreddit*[5].

In **Problem Set Y**, we challenge the user with a problem originally from the */r/dailyprogrammer subreddit*[6] that calculates the number of coins necessary to reach the desired amount. The function is given an ordered list of coins and the desired amount.

5.2 Result Analysis

In this section, we will analyze and discuss the collected background information from the participants in Subsection 5.2.1. Secondly, in Subsection 5.2.2, we will go over the performance of the participants during the experiment itself. These results are divided into four subsections: (1) the *Time to Reach Patch* meaning the time the participants take to find a valid patch for the tasks, (2) *Code* where we analyze the final code solutions submitted by the participants at the end of the experiment, (3) the *Survey* subsection which reflects the feedback the user gives, through the survey, in parallel with the practical experiments and finally, (4) in *Tool Usage*, how often the participants use our extension. Lastly, in Section 5.2.3 (p. 45), we outline the feedback from the participants after the completion of all the tasks in the practical experiment.

5.2.1 Background

To better get to know the participants of this experiment, a *Background* section with Likert-type statements are given to the volunteers before starting the experiment. This survey inquires the participants about their technical knowledge, their comfort with the experiment environment technologies, and their confidence in skills like debugging and problem-solving. This section is comprised of 8 questions denominated from B1 to B8.

B1: I have considerable experience with JavaScript;

B2: I have considerable experience with Testing Frameworks;

B3: I have considerable experience with Visual Studio Code (VSCode);

B4: I regularly use tools that offer suggestions to help me code (linters, code completion, etc);

B5: I feel comfortable understanding code I have not seen before;

B6: I feel comfortable in identifying bugs in code I have not seen before;

B7: I feel comfortable in fixing bugs in code I have not seen before;

[5] https://www.reddit.com/r/dailyprogrammer/comments/55nior/20161003_challenge_286_easy_reverse_factorial/ Retrieved on: 29/06/2021

[6] https://www.reddit.com/r/dailyprogrammer/comments/7ttiq5/20180129_challenge_349_easy_change_calculator/ Retrieved on: 29/06/2021

B8: I feel comfortable in optimizing code I have not seen before.

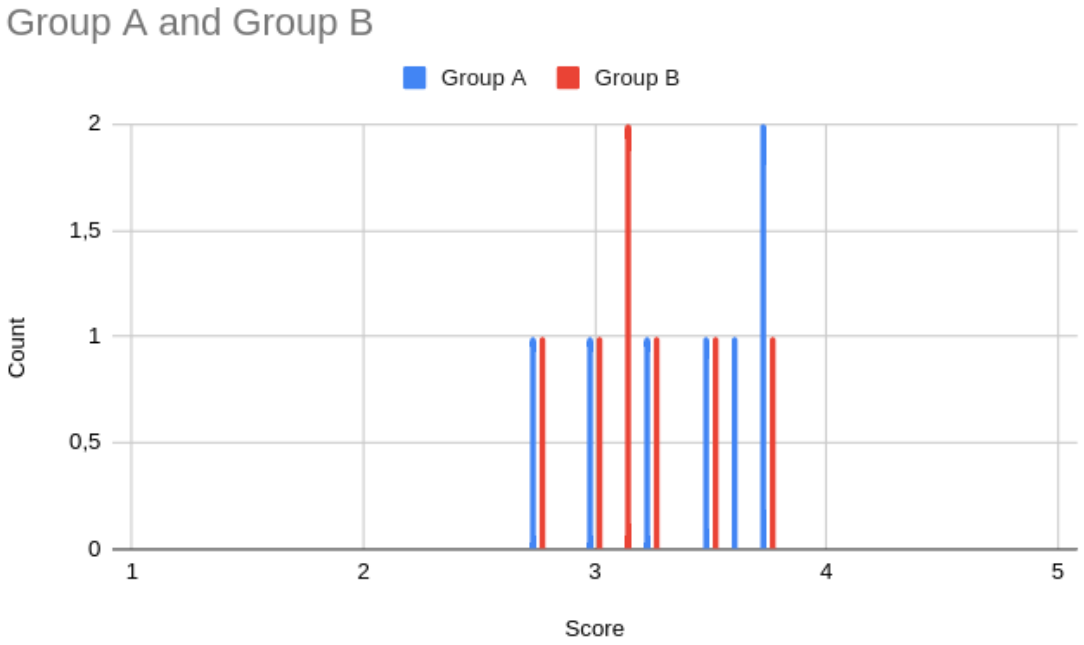

Figure 5.2: Histogram of the scores for the *Background* survey.

To calculate the score for each participant, we use the average of the score for each item where each statement is rated on a scale of 1 (*Strongly Disagree* through 5 (*Strongly Agree*). Consequently, the minimum and maximum values are 1 and 5, respectively, and the average is 3.

To analyze the scores of both groups, we plot the scores of the participants in Figure 5.2. Despite the values showing a tendency for scores slightly above the average, both groups seem to follow this trend meaning there is no apparent significant difference between the groups' technical knowledge in this histogram.

Table 5.1: Statistical values and $p-values$ for hypothesis testing the answers from the *Background* survey.

Group	Size	Mean	Std. Deviation	Shapiro-Wilk (p)	Levene (p)	t-test (p)
A	7	3.37	0.38	0.33	0.24	0.41
B	7	3.21	0.32	0.89		

To further study potential background differences between the groups and check if there is any evidence that they are statistically different, we test the null hypothesis for both groups being sampled from a population with an equal mean.

As seen in Table 5.1, both groups have a higher mean than the average of 3. Furthermore, and to evaluate if there is any statistically relevant difference between the two groups, we use *Levene's test for equality of variances*, the *Shapiro-Wilk* test, and finally, the *Student's t-test*.

For the *Levene's test for equality of variances*, we test for the null hypothesis that both groups derive from a population with equal variances. After finding the *p-value* of 0.24, which is substantially bigger than the significance level of *0.05*, we **cannot reject the null hypothesis**.

Secondly, we test the null hypothesis to understand if the groups derive from a population with a normal distribution. With this in mind, we use the *Shapiro-Wilk* test. The resulting *p-value* from this test is also significantly bigger than the significance level of *0.05*, and therefore we **cannot reject the null hypothesis**.

Finally, we use the *Student's t-test* to test the null hypothesis that the means of the populations from which group are sampled are the same. After obtaining the *p-value* of 0.41, which is significantly bigger than the significance level of *0.05*, **we cannot reject the null hypothesis**. We thus can conclude that there is no statistical difference between the groups.

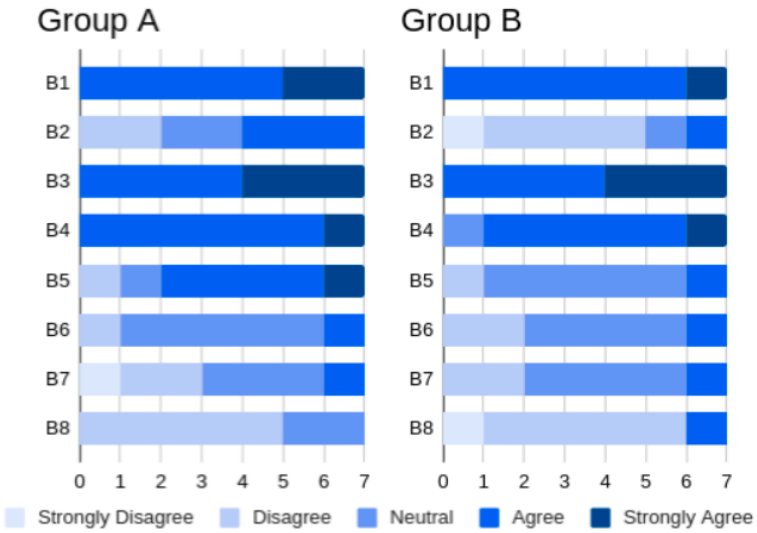

Figure 5.3: Stacked Bar Chart with the frequencies for Group A (left) and Group B (right) for the answers of the *Background* survey section.

Moreover, we can analyze the specific answers for each item to find interesting trends. Figure 5.3 shows, for each *Background* item, the frequency of each answer for each group. Items **B1**, **B3** and **B4** show that both groups are generally comfortable with the environment used in the experiments by agreeing they were familiarized using *JavaScript* (**B1**), *VSCode* (**B3**) and *external tools that offer suggestions* (**B4**):

B1: I have considerable experience with JavaScript;

B3: I have considerable experience with Visual Studio Code (VSCode);

B4: I regularly use tools that offer suggestions to help me code (linters, code completion, etc).

On the other hand, the users disagree the most with items **B2**, **B8** and, to a lower extent, **B7** which show a **lack of experience using testing frameworks** as well as some **unease to fix bugs or optimize code** never seen before.

B2: I have considerable experience with Testing Frameworks;

B7: I feel comfortable in fixing bugs in code I have not seen before;

B8: I feel comfortable in optimizing code I have not seen before.

5.2.2 Practical Experiment

For this empirical work, the volunteers are required to solve 2 *ProblemSets* with 4 *Tasks* each. To measure their performance during these tasks, and as mentioned in Subsection 5.1.1, three metrics are recorded:

- the time to reach the first patch;
- the code of the final solution;
- whether or not a suggestion was accepted.

5.2.2.1 Time to Reach Patch

The Table 5.2 shows the *Mean* and *StandardDeviation* for each task and group. Except for *Task 4* from *Problem Set X*, we can see that every mean for the time to reach a patch with the use of the extension is smaller than without its use which would hint that, by using the tool, in average, a user is faster in completing the assignment. Furthermore, tasks 2 and 3 show the biggest difference in their means, which is according to our expectations since the users show a lack of confidence in their skills to optimize unseen code in the answers to *Background* question **B8** (*cf.* Subsection 5.2.1, p. 36) and, with the use of the tool, a suggestion would be immediate.

However, to evaluate if the difference in time to reach patch by using the tool is statistically different, we decide to do a two-sided *Mann-Whiney U test* for each task to test if the time spent when using the extension is significantly higher or lower than without its use. With these tests, we want to assess the following hypothesis:

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.

According to the values in Table 5.2, most *p-values* of the *Mann-Whiney U test* are smaller than the standard threshold of *0.05* defined. These underlined values for tasks 2 and 3 from both problem sets and task 1 from problem set X as well as task 4 from problem set y show that we can reject the Null Hypothesis that the distributions of the populations are identical for these tasks. On the other, for the remainder of the assignments, we **cannot reject the Null Hypothesis**. Therefore we are not able to confirm any statistical evidence that the groups performed differently.

Table 5.2: Statistical values and the *Mann-Whiney U p-value* for the times to reach patch for each group and task

Task	Set	Tool	Mean	Std. Deviation	Mann-Whiney U p-value
1	X	Yes	02:07	00:23	0,00058
		No	03:19	00:17	
	Y	Yes	01:26	00:41	0,14130
		No	01:54	00:35	
2	X	Yes	01:47	00:25	0,00408
		No	03:48	01:06	
	Y	Yes	01:55	00:27	0,01748
		No	02:49	00:44	
3	X	Yes	01:41	00:26	0,00233
		No	03:45	01:28	
	Y	Yes	01:24	00:38	0,00699
		No	02:41	00:38	
4	X	Yes	06:22	00:33	0,24810
		No	05:24	01:28	
	Y	Yes	03:35	00:38	0,00116
		No	05:50	01:27	

5.2.2.2 Code

Although the time to reach a patch is a significant metric to analyze the effects of the tool in a participant's workflow, we ought to understand how the suggestions from the extension may affect the quality and readability of the code. Considering the inherent freedom one has when asked to optimize a program on a metric other than its correctness (program repair), the participants' solutions show significantly different patches for these two types of tasks.

Program Repair: In *Task 4* of *Problem Set Y*, after the user follows the instructions to update the value of $map.total$, the program still has a bug in an if condition. The bug happens because the program continues to add coins while the total is less than the wanted instead of while the total is less or equal to consider the final iteration. The bug is found in the following condition:

```
1    // From Problem Set Y, Task 4, line 22
2    if (coin + map.coinTotal < change)
```

The extension suggests four different possible patches:

```
1    // Patch 1
2    if ((coin - 1) + map.coinTotal < change)
3    // Patch 2
4    if (coin + (map.coinTotal - 1) < change)
5    // Patch 3
6    if (coin + map.coinTotal <= change)
7    // Patch 4
8    if (coin + map.coinTotal < (change + 1))
```

Despite each patch being semantically equivalent, one might claim that the 3rd Patch is a better portrait of what the condition is meant for and thus a better solution. When analyzing the participants' solutions, we find that **from all the participants that do not use the extension, every participant fixes the program using Patch 3. In contrast, participants using our tool select the first patch on the suggestion list, the Patch 1**, except for one user that finished the Task by using the Patch 3;

Program Optimization: Within program optimization, participants show a clear distinction between patches accepted from the extension and human-generated patches.

Despite also following the same logic of minimizing the number of operations made, users **tend not to introduce new constant variables**. Instead, they prefer rewriting the formulas when not using our tool.

For example, in *Task 3* of *Problem Set X*, for the following code snippet:

```
1    // From Problem Set X, Task 3
2    for(var t = 0; t < interval; t++) {
3        var newY = posY + velY*t - (9.8/2)*t*t;
4        answer.push(newY);
5    }
```

When using the tool, every user accepts the suggestion made to simplify the constant value from `(9.8/2)` to `(4.9)`. This pattern is also seen in the analogous task from *Problem Set Y* where developers simplify `(Math.Pi/3)` to `(1.0471975511965976)` despite their behavior showing some hesitance and even having 2 cases where a new constant variable is made to explain the meaning of this number.

On the other hand, when not using the tool, 6 out of the 7 users decide that the better solution to minimize the number of operations in this loop is to to rearrange the operations like in the following example:

```
1    // From Problem Set X, Task 3
2    for(var t = 0; t < interval; t++) {
3        var newY = posY + ((velY - (9.8/2)*t))*t;
4        answer.push(newY);
5    }
```

One other interesting sight was that, in *Group A*, which does not have access to the tool in the first half of the experiment, **only one user out of seven substitutes one of the keywords** with the intent to improve the performance of the program. On the other hand, in *Group B*, the group which does have access to the tool in the first half, **every user changes at least one instance of a keyword** for another one in the hopes to optimize the code, which hints to the fact that the users from *Group B* learn that these substitutions are a possible way to optimize the code.

5.2.2.3 Survey

After solving each *Problem Set*, the participants are asked to fill a survey about the freshly done tasks to get a feel of their immediate feedback.

P1: The bugs were easy to identify;

P2: The solutions were straightforward;

P3: I feel like I spent more time in identifying the bugs than in solving them;

P4: I was able to confidently make improvements in the code;

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

P6: I used the fixes suggested by the extension. (correctness only, not regarding optimization suggestions);

P7: I understood the fixes suggested by the extension;

P8: I used the improvement suggestions by the tool;

P9: I tried to understand the improvements suggested before accepting them;

P10: The difficulty of the problems were similar.

Depending on whether the user has access to our tool or not, and depending on the *Problem Set*, the survey is slightly different to accommodate for the different environment and context.

Items from *P*1 through *P*4 are asked in every stage since they are only meant to judge the users' perspectives on the tasks. However, in sections where the user has access to the extension, they are also presented with *P*5 through *P*9. Finally, *P*10 is available at the beginning of the second half of the survey to minimize the time from completing the tasks to answering this item and thus have the best possible comparison.

Problem Set X: As seen in Figure 5.4, there are no significant differences between the answers in common for both groups. There is, however, a small skew to the right for the *Group B* which is the group using the tool in this stage. This skew may be influenced by the tool's availability, making the problems seem easier when solving them with the extension. Moreover, one may argue the skew on *P4* is slightly bigger, which, as long as the users trust the tool, would be according to expectations since there is considerable unease in optimizing code (*cf.* Subsection 5.2.1, p. 36).

Regarding the extension-specific questions, the users show very positive reactions regarding the extension speed in identifying fixes (**P5**), usage, and attempt to understand the proposed patches (**P6** and **PX7**) and the adhesion of the tool for optimization purposes (**P8**). There is, however, more receptive feedback from the user regarding understanding the improvements, this might be because the suggestions went against some coding guidelines the user follows (*cf.* Subsection 5.2.2.2, p. 40) or maybe, the user does not understand the reason why the patch is considered to have better metrics;

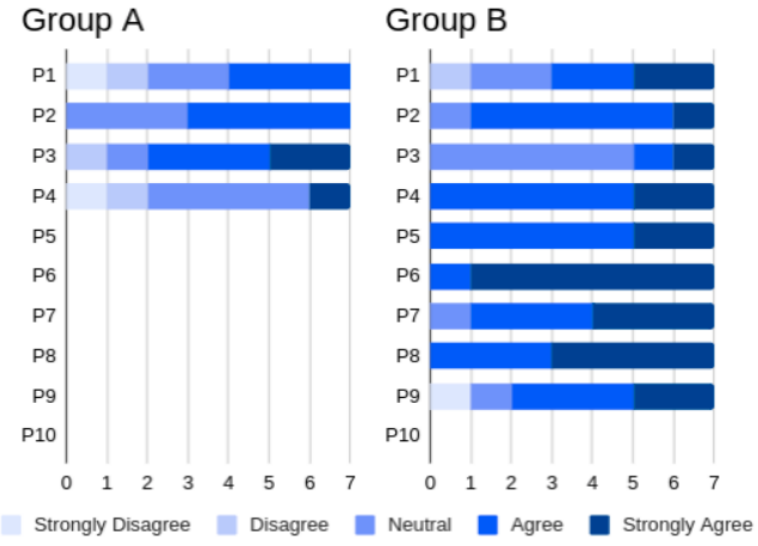

Figure 5.4: Stacked Bar Chart with the frequencies for Group A (left) and Group B (right) for the answers of the survey section regarding *Problem Set X*

Problem Set Y: As for *Problem Set Y*, as seen in Figure 5.5, three of the first four items in *Group B*, **P1**, **P2** and **P4**, show a considerable skew to the left suggesting that members of *Group B* found the bugs from this set harder to identify, with not so straightforward solutions and had

overall less confidence in their improvements. This is counter-intuitive since we expect the group to have learned some tips and tricks from the previous problem set. However, there is a strong similarity between the results from both groups when not using the tool, which further hints that the problems are perceived to be easier when using the tool.

The results from *Group A* are also positive when using the extension. Moreover, this group is more critical to accept suggestions from the tool without understanding them. We believe this is due to some intellectual curiosity this group might have from finally discovering if their performance in the previous section (in *Problem Set X*) is up to what was expected or not by critically comparing the suggestions to their previous improvements.

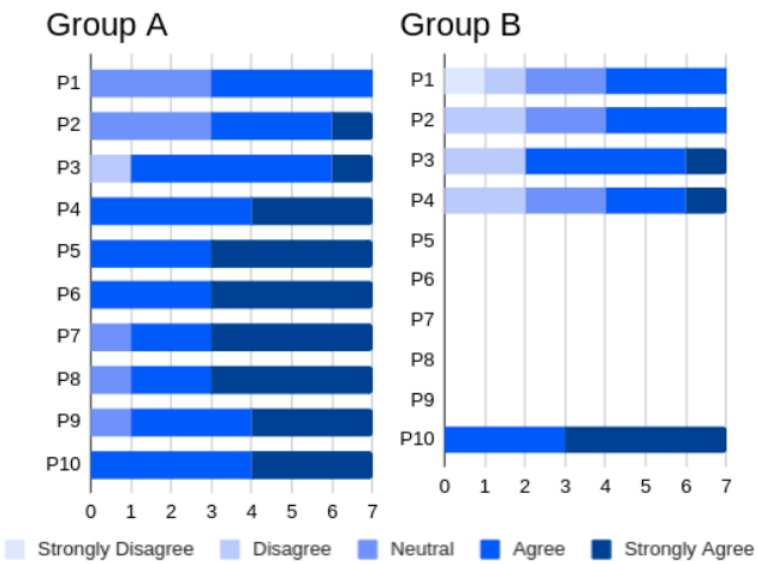

Figure 5.5: Stacked Bar Chart with the frequencies for Group A (left) and Group B (right) for the answers of the survey section regarding *Problem Set Y*

Finally, according to **P10**, and to make sure these results are comparable, we ask the participants if they feel that the analogous problems from the different problem sets are similar, which the users strongly agree with.

5.2.2.4 Usage of Extension

During the experiment stage, in which the participants can use the extension, the user has to choose to either accept or not the suggestion. However, for both of *Tasks 4*, since bugs are only detected by the tool after everything else is correct, if a user corrects the present bug before finishing the implementation, the extension does not give any suggestion throughout the task. Nonetheless, for each task from each problem set, we record if the user accepts a suggestion or not. A suggestion is considered accepted if the participant uses the native *VSCode Quick Fix* option or if the user, after reading the suggestions given, opts into using the concept of one of the patches suggested.

Table 5.3: Usage of the tool for each *Task* and *Problem Set*

Task	Problem Set	Usage (%)
1	X	85.7
	Y	71.4
2	X	100.0
	Y	100.0
3	X	100.0
	Y	100.0
4	X	71.4
	Y	100.0

To understand the results from the Table 5.3, we need to understand the characteristics of the tasks and how they may influence the results. Considering all of the tasks 2 and 3 refer to the optimization of code after it being semantically correct, something found in Section 5.2.1 (p. 36) that participants are not too comfortable with, all of the users accept the immediate suggestion provided by our tool. The only variations seen are from users on *Task 3* of the *Problem Set Y* who, after getting the suggestion of substituting `Math.PI/3` for `1.0471975511965976`, decide to create a variable with this constant to clarify its meaning.

For *Tasks 1*, there is a slightly lower number for *Problem Set Y* which might be evidence that two of the seven users from *Group A*, after debugging and solving an identical problem in the previous problem set are expecting a similar solution.

Finally, by comparing the two results from *Task 4*, it seems that the extension does better with the assignment from *Problem Set X*. However, the 0,29% of participants that do not use the extension in this task, fix the bug before finishing implementing the function. We believe that this may have happened because the bug present in the first set is more likely to catch a developer's attention than its analogous since the remainder of an operation is usually not negative, especially in the context of factorial numbers, which are usually positive.

```
1 // Bug found on Task 4, Problem Set X
2 if (num % cand < 0)
3 // Bug found on Task 4, Problem Set Y
4 if (coin + map.coinTotal < change)
```

5.2.3 Post-Test Survey

After all the tasks are completed, the users are asked to fill a final survey, the *Post-Test* survey. With this survey, we want to understand the overall perspective of the participants towards the effect of our tool during the experiment. The *Post-Test* had the following 5 statements:

PT1: The tool was simple to use;

PT2: This tool can positively impact my development workflow;

PT3: I would consider using this tool;

PT4: A tool like this is likely to distract me from my development;

PT5: I would trust the tool.

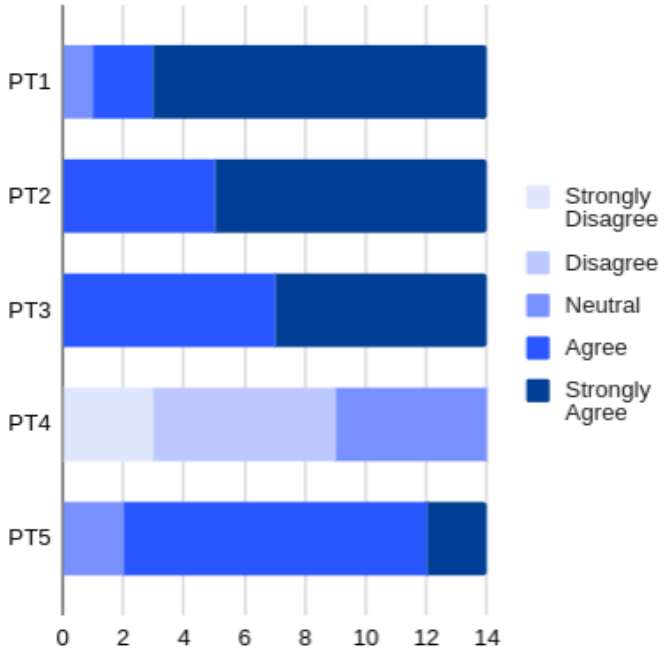

Figure 5.6: Stacked Bar Chart with the frequencies for the answers of the survey section regarding the *Post Test*.

The participants are in great agreement with the statements **PT1**, **PT2**, **PT3** and **PT5** indicating that the tool is simple to use, could positively impact developers' workflow. Moreover, users agree they would consider using the tool and that they trust the tool output.

5.3 Main Findings

With the experiments finished and the results recorded, we expect to answer our *Research Questions* previously made in Section 3.2 (p. 18):

RQ1: *Are users faster in reaching a patch when using a real-time Automatic Program Optimization tool?*

On the one hand, when comparing the time to reach a patch in Subsection 5.2.2.1, we note that for tasks 2 and 3, where a user is asked to optimize a program in a metric other than

program correctness, we can reject the null hypothesis that there is no statistical difference between the results seen. Therefore there is evidence supporting our assumptions regarding **RQ1** for optimizations on energy and speed.

On the other hand, tasks 1 and 4, specifically meant for optimizing the program correctness, do not have conclusive results. For tasks 1, we believe the discrepancy seen is caused by the similarity and simplicity of the two tasks, which may lead the developers to learn the solution during the first section (in *Problem Set X*) and then be considerably faster, and thus more consistent, in the task of the *Problem Set Y*. On both tasks 4, we find that developers spend more time understanding the rationale behind the recursive call than fixing the underlying bug. Conversely, for task 4 in *Problem Set Y*, a direct instruction to update a variable was given, and the core rationale of the algorithm is already written.

Nevertheless, the observed differences of the means between analogous tasks when using our tool and when not using it are significant. We see that times to reach a patch with our extension are consistently lower than its counterpart in 75% of our assignments. For that reason, we argue that there is significant statistical evidence to support that **users are faster at reaching a patch** when using an Automatic Program Optimization tool.

RQ2: *Are users aware of the rationale suggestions generated by an Automatic Program Optimization tool before accepting them?*

Regarding **RQ2**, we ask our participants, after answering each problem set using our tool, if they understand the suggestion given. The answers are overwhelmingly positive for both *Group A* and *Group B* despite being more so for *Group A*, which we suspect is caused by some curiosity one might have to compare their previous solutions without the extension with the patches suggested by our tool.

However, the final solutions are not as conclusive. Despite their final code for Tasks 2 and 3 supporting our claim since every user on *Problem Set Y* from *Group B*, who has access to the extension in the first stage, uses an identical mutation learned in the previous problem set as observed in Subsection 5.2.2.2. When faced with a problem with a different rationale, like Task 4 from *Problem Set Y*, participants using our extension overwhelmingly accept patches that are deemed not as human friendly, which suggests that our survey answers may be skewed by the willingness of the participants to agree with what they may think is our hypothesis.

While it is possible that despite the users not accepting what we consider is the most human-friendly option and thus consistent with the survey feedback, we believe that there is enough evidence suggesting otherwise. Therefore, **it's not conclusive** that the users understand the rationale behind the suggestions made by the tool.

RQ3: *Are solutions programmed by human developers different from the solutions generated by an Automatic Program Optimization tool?*

As seen in Subsection 5.2.2.2, there is a consistent difference between the patch found in Task 4 from *Problem Set Y* between participants using the tool and those who do not. Related to our **RQ2**, we believe that users with the tool, by not completely understanding the rationale behind the suggestion, steer towards picking the first seen suggested patch rather than choosing the alternative that most closely resembles what we consider to be the best natural language representation of the condition. Hence, we find that solutions generated by an Automatic Program Optimization **are mostly different**.

5.4 Threats to Validity

Every empirical study suffers from threats to validity that must be studied and considered when defining the structure and environment of the experiment and how to interpret the results critically. For each of the following validity threats: *Construct Validity*, *Internal Validity* and *External Validity*, we analyze what they are and how they may affect our experiment.

5.4.1 Construct Validity

Construct validity refers to how well our assumptions and overall construction of the environment were made. In this study, it would include the tasks, environment, and collected data.

Validity of the Problem Sets

To assure the fidelity of the experiment, the problems must represent realistic problems that a developer is likely to face during their professional work. To mitigate this threat, we use either existing code the program repair tasks and try to simplify optimization problems to make them manageable for our time constraints and context.

Furthermore, the assumptions made from comparing the results from analogous tasks from different problem sets assume that the problems are of similar difficulty. To understand if this would be a serious factor, we ask our users what their perception is after finishing all the tasks. The participants' responses show that the problems from the different sets are not significantly different as seen in the answers to **P10** in Subsection 5.2.2.3.

Hypothesis Guessing

As a threat to the quality of our tool usage metrics, it is possible that users do not just passively participate in the experiment but also are actively trying to guess our hypothesis. This may have resulted in higher values of tool usage percentage since it is the most notable difference in the environment between the two stages of the experiment.

Environment and Data Collecting

Despite the environment being constant since every experiment is done remotely on the same machine, the variable delay in the remote connection may have some impact on users' rhythm and performance. Furthermore, our definition of *a valid patch* is of a patch that either improves the program correctness in the APR tasks or introduces optimization for

tasks 2 or 3. Since we are not studying the objective improvement in optimization, this process becomes somewhat subjective. Consequently, the times to reach the patch were recorded by observation which may introduce some error in the collected data.

5.4.2 Internal Validity

Internal validity pertains to the security in inferring conclusions from a set of independent variables and the effects observed.

Motivation of the Participants

In these experiments, we try to study if the usage of our tool impacted the time a user takes to reach a patch. However, since the tasks are comprised of a single-function problem and there is no concept of ownership or maintenance (i.e., the user is not expected to maintain this codebase in the future), one can hypothesize that the patches found are not sustainable and therefore would not be found in a professional setting.

Physical Environment

As a consequence of the experiments being fully remote and the user not being required to have nor a microphone nor a camera turned on at all times, the control of the environment not only is impossible to enforce but is harder to record. Our strategy to mitigate this factor is to inform the users, before the experiment, that they should be alone in a room for 30 to 40 minutes to be considered available for this experiment.

5.4.3 External Validity

Finally, external validity reflects how well the findings of a study can be generalized to different samples of participants and settings.

Sample Size

There were 14 participants in this study's experiment, divided into two groups. In the future, this should be taken into account, and more users should be tested for the mean values to be more accurate and to decrease the impact of possible outliers in the study.

5.5 Summary

In Section 5.1 (p. 31), we start off by outlining the process in which the experiments were conducted, including the ideal participant profile and the environment in which the tests take place. We further detail the tasks included in the experiment and their characteristics.

In Section 5.2 (p. 36), we analyze the answers from the surveys taken by the participants, the characteristics of their final solutions to our tasks, and finally, we discuss how these findings fit in our Research Questions. Lastly, in Section 5.4 (p. 48), we go over the main threats to the validity of our study and experiments and the steps taken to mitigate them.

Chapter 6

Conclusions

6.1 Conclusions

In this work, we explore the effects of the usage of an Automated Program Optimization tool during the workflow of a developer in order to test our hypothesis:

> *"Using a real-time Automatic Program Optimization tool improves the speed and final result of code solutions."*

To validate our hypothesis, we extended a *Visual Studio Code* created by *Campos* for [11] in order to not only assess the performance of developers when using a program-repair oriented extension but also for other, more generic, metrics that one might value. In this case, we used program repair, energy cost optimization, and performance optimization as our tested metrics. Afterward, we carried out an empirical study to evaluate the performance of software developers during their workflow in order to compare the performances both with the extension and without it. The aforementioned experiment was carried out to validate our hypothesis and answer our Research Questions:

RQ1: *Are users faster in reaching a patch when using a real-time Automatic Program Optimization tool?*

According to the study performed, a user was faster to reach a patch with the extension versus its counterpart in 75% of the assignments that we tested the participants on. Therefore, we claim that **users are faster to reach a patch** when using a real-time Automatic Program Optimization tool.

RQ2: *Are users aware of the rationale suggestions generated by an Automatic Program Optimization tool before accepting them?*

Due to inconsistencies between the users' feedback through the surveys, where participants strongly agreed that they understood the suggestions, their actual behavior, and their final code solutions, we believe users may have tried guessing our hypothesis when answering

the surveys. Findings in the final code solutions suggest that participants without the extension find solutions that best describe the solution in natural language whereas we believe most users with the extension accept suggestions as long as they are semantically correct. Therefore, we believe our findings **were not conclusive**.

RQ3: *Are solutions programmed by human developers different from the solutions generated by an Automatic Program Optimization tool?*

Although we do not claim the energy and speed optimization mutations are objectively the best (since they were generated from trial and error via mutations), we found evidence that solutions generated by our tool are **significantly different** from solutions proposed by our participants. Furthermore, we argue that the natural language meaning of the solutions suggested by our tool is often very different from the human-written solutions which might be an indicator that the solutions are not as easy to interpret.

To sum up, we claim that the usage of an Automated Program Optimization tool **significantly decreases the time to find a patch**. Furthermore, we found evidence that solutions generated by our tool were **different from the solutions generated by our participants** when not using the extension. However, since this experiment was only partaken by 14 participants, may not portrait the reality and in the future new experiments should be made with this in mind.

6.2 Main Contributions

The main contributions of this work can be summarized in the following two items:

- A Visual Studio Code implementing a real-time Automated Program Optimization tool, which is composed of three main components that are easily extendable to increase the type of operators used, the strategy in which the extension chooses what operators to use and which mutations to accept and finally, environments that can be defined to evaluate in either run-time or statically the score of a specific mutation;
- An **empirical study** with 14 participants to assess the usefulness of the extension developed by dividing the participants into two groups which had to finish a total of 8 tasks. The tasks themselves were divided into two problem sets, with analogous tasks of similar difficulty, in which we could further evaluate the impact of the tool.

6.3 Future Work

Despite the extension being ready to be extended with the addition of new mutation operators, new strategies, and new environments to evaluate new metrics, there are still features that can be implemented and tested for:

- Creating another component directed towards bottleneck localization would decrease the number of positions in which the mutations are applied and consequently increase the number of mutations that can be applied without losing the real-time aspect of the suggestions;
- As seen in this work, one of the struggles this technology may face in future adoption relies on developers not understanding the meaning of the suggestion. Therefore, we believe there is great value in creating a system that is coupled with the operators that may provide some insight on why the mutation might have better characteristics;
- Organize more experiments taking into account the recommendations given previously in Section 5.4 (p. 48) to further explore how Automated Program Optimization can be used in the development workflow.
- Finally, we believe that an Automated Program Optimization tool should be able to give a confidence level on the suggestion it gives to the user to help the developer make their decision.

Appendix A

Code from Experiments

This appendix is divided into 4 sections, one for each pair of Tasks. For each Task, two listings can be found: the first for *Problem Set X* and the second for *Problem Set Y*.

A.1 Task 1

A.1.1 Problem Set X

```
let assert = require('assert')

/*
Problem 1: Substring

Given a string a and two indexes, find the substring contained within those two
    indexes.

The usage of your function should be:

mySubstring('Mozilla', 1, 2) = 'oz'
mySubstring('Mozilla', 3, 3) = 'i'

Use the JavaScript substring function (usage: stringVariable.substring(num1, num2))
    .
*/

function mySubstring(str, i1, i2) {
    return str
}

describe('mySubstring', function() {
    it('should return a substring. #fix {mySubstring} (1)', function() {
        assert.strictEqual(mySubstring('This is a string.', 1, 2), 'hi')
    })
```

```
24
25      it('should return a substring. #fix {mySubstring} (2)', function() {
26          assert.strictEqual(mySubstring('This is a string.', 6, 8), 's a')
27      })
28  })
```

A.1.2 Problem Set Y

```
1   let assert = require('assert')
2
3   /*
4
5   Problem 1: Splice
6
7   Given an array a and two indexes, find the slice contained within those two indexes
        (both included).
8
9   The usage of your function should be:
10
11  mySlice(['a', 'b', 'c', 'd'], 1, 2) = ['b', 'c']
12  mySlice(['a', 'b', 'c', 'd'], 3, 3) = ['d']
13
14  Use the JavaScript Slice function (usage: arrayVariable.slice(num1, num2)).
15  */
16
17  function mySlice(arr, i1, i2) {
18      return arr
19  }
20
21  describe('mySlice', function() {
22      it('should return reverse factorial. #fix {mySlice} (1)', function() {
23          assert.deepStrictEqual(mySlice(['a', 'b', 'c', 'd'], 1, 2), ['b', 'c'])
24      })
25
26      it('should return reverse factorial. #fix {mySlice} (2)', function() {
27          assert.deepStrictEqual(mySlice(['a', 'b', 'c', 'd'], 3, 3), ['d'])
28      })
29  })
```

A.2 Task 2

A.2.1 Problem Set X

```
var assert = require('assert')

/*

Problem 2: Optimize the bubble sort algorithm to make it run as fast as possible.
https://en.wikipedia.org/wiki/Bubble_sort
*/
function bubbleSort(inputArr) {
    let len = inputArr.length
    for (let i = 0; i < len; i++) {
        for (let j = 0; j != len; j++) {
            if (inputArr[j] > inputArr[j + 1]) {
                let tmp = inputArr[j]
                inputArr[j] = inputArr[j + 1]
                inputArr[j + 1] = tmp
            }
        }
    }
    return inputArr
}

describe('bubbleSort', function() {
    it('should return sorted array. #fix {bubbleSort} (1)', function() {
        assert.deepStrictEqual(bubbleSort([10]), [10])
    })

    it('should return sorted array. #fix {bubbleSort} (2)', function() {
        assert.deepStrictEqual(bubbleSort([1, 2, 3, 4]), [1, 2, 3, 4])
    })

    it('should return sorted array. #fix {bubbleSort} (3)', function() {
        assert.deepStrictEqual(bubbleSort([4, 2, 4, 1, 2, 2]), [1, 2, 2, 2, 4, 4])
    })
})
```

A.2.2 Problem Set Y

```
let assert = require('assert')

let gen = require('random-seed')

/*

Problem 2: Some Friends (a, b, c, d, e, ..., z) are playing a game and need to keep
    track of the scores.
Each time someone scores a point, the letter of his name is typed in lowercase.
```

```
If someone loses a point, the letter of his name is typed in uppercase.

Optimize the following algorithm to make it run as fast as possible.

abcde:
{
    a: 1
    b: 1
    c: 1
    d: 1
    e: 1
}

dbaCEDbacB:
{
    a: 2
    b: 1
    c: 0
    d: 0
    e: -1
}

*/


function tallyScore(n, seed) {
    let rand = gen.create(seed)
    let tally = {}

    for (let i = 0; i < n; i++) {
        let charCode = rand(58) + 65
        while (charCode >= 91 && charCode <= 96) {
            charCode = rand(58) + 65
        }
        assert(charCode >= 65 && charCode <= 122)
        let currentChar = String.fromCharCode(charCode)

        let lower = currentChar.toLowerCase()
        let point = tally[lower] || 0

        if (charCode >= 97) {
            tally[lower] = point + 1
        } else {
            tally[lower] = point - 1
        }
    }

    return tally
}
```

```

describe('tallyScore', function () {

    it('should return the score.. {tallyScore} (1)', function () {
        const result = {
            c: 195,
            v: -2,
            f: 179,
            s: 226,
            z: -289,
            h: -16,
            g: 75,
            y: -7,
            l: 182,
            w: 264,
            i: 194,
            j: -22,
            e: -413,
            t: -393,
            p: -330,
            u: 81,
            a: -245,
            n: -400,
            k: -91,
            b: -68,
            x: -157,
            r: -33,
            m: 5,
            q: 92,
            d: -77,
            o: 14
        };

        assert.deepStrictEqual(tallyScore(1000000, "first-test"), result);
    })
})
```

A.3 Task 3

A.3.1 Problem Set X

```
let assert = require('assert')

/*
Problem 2: The following function calculates the vertical position of a body after
    getting thrown in the air
```

```
    If possible, attempt to optimize this function so that it consumes the least
        amount of energy
*/

// A weight is thrown in the air, calculate its posY in the first 'interval' second
function positionCalculator(posY, velY, interval) {
    var answer = []

    for (var t = 0; t < interval; t++) {
        var newY = posY + velY * t - (9.8 / 2) * t * t;
        answer.push(newY);
    }

    return answer;
}

function areSimilar(preCalculated, newCalc) {
    assert.strictEqual(preCalculated.length, newCalc.length)
    for (let i = 0; i < newCalc.length; i++) {
        assert.ok(Math.abs(preCalculated[i] - newCalc[i]) < 0.001)
    }
}

describe('positionCalculator', function () {
    it('should return the position of the body. #fix {positionCalculator} (1)',
        function () {
        const preCalculated = [0, 95.1, 180.4, 255.9, 321.6, 377.5, 423.6, 459.9,
            486.4, 503.09999999999997, 510, 507.0999999999999, 494.4, 471.9,
            439.5999999999999, 397.5, 345.5999999999999, 283.89999999999986,
            212.39999999999986, 131.0999999999999, 40, -60.90000000000009,
            -171.60000000000036, -292.0999999999999, -422.4000000000001,
            -562.5000000000005, -712.4000000000001, -872.1000000000004,
            -1041.6000000000004, -1220.9000000000005, -1410, -1608.9000000000005,
            -1817.6000000000004, -2036.1000000000004, -2264.4000000000005, -2502.5,
             -2750.4000000000005, -3008.1000000000004, -3275.6000000000004,
            -3552.9000000000005, -3840, -4136.9, -4443.6, -4760.1,
            -5086.4000000000015, -5422.500000000002, -5768.4, -6124.1, -6489.6,
            -6864.9000000000015, -7250.000000000002, -7644.9, -8049.6,
            -8464.100000000002, -8888.400000000001, -9322.5, -9766.400000000001,
            -10220.1, -10683.600000000002, -11156.900000000001, -11640,
            -12132.900000000001, -12635.600000000002, -13148.100000000002,
            -13670.400000000001, -14202.5, -14744.400000000001,
            -15296.100000000002, -15857.600000000002, -16428.9, -17010, -17600.9,
            -18201.600000000002, -18812.100000000002, -19432.4, -20062.5, -20702.4,
             -21352.100000000002, -22011.600000000002, -22680.9, -23360, -24048.9,
            -24747.6, -25456.100000000006, -26174.4, -26902.500000000007, -27640.4,
             -28388.1, -29145.600000000006, -29912.9, -30690.000000000007,
            -31476.9, -32273.6, -33080.100000000006, -33896.4, -34722.50000000001,
            -35558.4, -36404.1, -37259.600000000006, -38124.9];
```

```
30          const newCalc = positionCalculator(0, 100, 100)
31          areSimilar(preCalculated, newCalc)
32      })
33
34      it('should return the position of the body. #fix {positionCalculator} (2)',
            function () {
35          const preCalculated = [0, 45.1, 80.4, 105.9, 121.6, 127.5, 123.6,
                109.89999999999998, 86.39999999999998, 53.099999999999966, 10,
                -42.90000000000009, -105.60000000000002, -178.10000000000002,
                -260.4000000000001, -352.5, -454.4000000000001, -566.1000000000001,
                -687.6000000000001, -818.9000000000001, -960, -1110.9,
                -1271.6000000000004, -1442.1, -1622.4, -1812.5000000000005, -2012.4,
                -2222.1000000000004, -2441.6000000000004, -2670.9000000000005, -2910,
                -3158.9000000000005, -3417.6000000000004, -3686.1000000000004,
                -3964.4000000000005, -4252.5, -4550.400000000001, -4858.1, -5175.6,
                -5502.900000000001, -5840, -6186.9, -6543.6, -6910.1,
                -7286.4000000000015, -7672.500000000002, -8068.4, -8474.1, -8889.6,
                -9314.900000000001, -9750.000000000002, -10194.9, -10649.6,
                -11114.100000000002, -11588.400000000001, -12072.5,
                -12566.400000000001, -13070.1, -13583.600000000002,
                -14106.900000000001, -14640, -15182.900000000001, -15735.600000000002,
                -16298.100000000002, -16870.4, -17452.5, -18044.4, -18646.100000000002,
                 -19257.600000000002, -19878.9, -20510, -21150.9, -21801.600000000002,
                -22462.100000000002, -23132.4, -23812.5, -24502.4, -25202.100000000002,
                 -25911.600000000002, -26630.9, -27360, -28098.9, -28847.6,
                -29606.100000000006, -30374.4, -31152.500000000007, -31940.4, -32738.1,
                 -33545.600000000006, -34362.9];
36          const newCalc = positionCalculator(0, 50, 90)
37          areSimilar(preCalculated, newCalc)
38      })
39  })
```

A.3.2 Problem Set Y

```
1   let assert = require('assert')
2
3   /*
4   Problem 3: The following function calculates the volume of Gabriel's Horn, decribed
        as the circular volume following 1/x from 1 to infinity
5       If possible, attempt to optimize this function so that it consumes the least
            amount of energy
6   */
7
8   function gabrielHornVolume(delta) {
9
10      var INF = 1e5;
```

```
    var volume = 0;
    for(var i = 1.0; i < INF; i += delta) {
        var h1 = 1.0/i;
        var h2 = 1.0/(i+delta)
        volume += Math.PI/3*delta*(h1*h1+h1*h2+h2*h2)
    }

    console.log(volume);
    return volume
}

describe('gabrielHornVolume', function() {
    it('should return the volume of the horn. #fix {gabrielHornVolume} (1)',
        function() {
        const volume1 = gabrielHornVolume(0.01)
        assert.ok(Math.abs(Math.PI-volume1) < 0.001)

        const volume2 = gabrielHornVolume(0.009)
        assert.ok(Math.abs(Math.PI-volume2) < 0.001)
    })
})
```

A.4 Task 4

A.4.1 Problem Set X

```
let assert = require('assert')

/*

Problem 4: Reverse Factorial

Everyone knows 5! corresponds to 5 * 4 * 3 * 2 * 1 = 120. In this problem, you need
     to write a function which, given
120, returns 5 - the reverse factorial.

If there is no number possible, return -1.

Hint: The strategy is pretty straightforward, just divide the term "num" by
    successively larger terms until you get to "1" as the resultant:
*/

function unfactorial(num, cand = 1) {
    let isCand = num === cand
  if (isCand) return cand
  if (num % cand < 0) return -1
```

```
19
20      // Implement Recursive call
21  }
22
23  describe('unfactorial', function() {
24      it('should return reverse factorial. #fix {unfactorial} (1)', function() {
25          assert.equal(unfactorial(120), 5)
26      })
27
28      it('should return reverse factorial. #fix {unfactorial} (2)', function() {
29          assert.equal(unfactorial(150), -1)
30      })
31  })
```

A.4.2 Problem Set Y

```
1   let assert = require('assert')
2
3   /*
4
5   Problem 4: Finish this partial implementation of a change processing function.
6
7   As an input, you're given an array with the coins you have, as well as the desired
        change to give. You must
8   output the smallest number of coins possible to return the correct value of change.
9
10  Assume the array is ordered from largest to smallest.
11
12  */
13
14  function processChange(arr, change) {
15    if (arr.length === 0) {
16      return -1
17    }
18
19    let result = arr.reduce(
20      (map, num) => {
21        let coin = parseInt(num)
22        if (coin + map.coinTotal < change) {
23          map.coinTotal += coin
24          map.coins.push(coin)
25          // Update the "total" value
26        }
27        return map
28      },
29      { total: 0, coinTotal: 0, coins: [] }
```

```
  )

  arr.splice(0, 1)

  return result.coinTotal === change ? result.total : processChange(arr, change)
}

describe('processChange', function () {
  it('should return change. #fix {processChange} (1)', function () {
    assert.equal(processChange([100, 50, 50, 50, 50], 150), 2)
  })

  it('should return change. #fix {processChange} (2)', function () {
    assert.equal(processChange([2, 4, 5], 10), -1)
  })
})
```

Appendix B

Raw Data Collected

This appendix comprises the information collected during the practical experiments. Table B.1 and Table B.2, for *Group A* and *Group B* respectively, display the information for each task from *Problem Set X* and *Problem Set Y* for each participant. Participants are represented by the concatenation of their assigned group, *A* or *B*, and their chronological order, 1 through 7 (i.e. A3 was the third participant from *Group A*.)

Table B.1: Raw Data Collected from Group A experiments. Values with an asterisk represents users that do not use the tool, only applicable to the section in which the user has access to the tool.

Group A		A1	A2	A3	A4	A5	A6	A7
Problem Set X (without tool)	Task 1	190	217	166	216	208	197	200
	Task 2	230	320	223	287	189	238	115
	Task 3	133	369	307	128	195	243	200
	Task 4	357	420	291	165	420	323	292
Problem Set Y (with tool)	Task 1	75	97	174	65	58	83	54
	Task 2	80	161	139	118	99	108	100
	Task 3	59	103	36	68	156*	91	75
	Task 4	234	225	235*	169	277	191	230

Table B.2: Raw Data Collected from Group B experiments. Values with an asterisk represents users that do not use the tool, only applicable to the section in which the user has access to the tool.

Group B		B1	B2	B3	B4	B5	B6	B7
Problem Set X (with tool)	Task 1	95	136	152	111	128	109	159
	Task 2	143	125	104	72	118	77	111
	Task 3	79	155	96	109	99	100	75
	Task 4	334	377	418*	406*	348	420*	371
Problem Set Y (without tool)	Task 1	102	57	109	164	97	153	117
	Task 2	134	128	259	148	170	158	190
	Task 3	168	161	243	143	140	130	148
	Task 4	279	295	337	269	370	521	382

References

[1] Developer Mozilla reference statements declarations in javascript. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements#declarations. Accessed: 2020-06-29.

[2] Rapl on linux. http://web.eece.maine.edu/~vweaver/projects/rapl/. Accessed: 2021-02-07.

[3] Abstract the economics of software maintenance in the twenty first century, 2006.

[4] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: Tightening the feedback loops. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, Programming '19, New York, NY, USA, 2019. Association for Computing Machinery.

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

[6] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[7] Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., USA, 1982.

[8] Alan W. Biermann. Approaches to automatic programming. volume 15 of *Advances in Computers*, pages 1–63. Elsevier, 1976.

[9] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. Pharo by example. square bracket associates, 2009. *URL http://pharobyexample. org*.

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

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

[12] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194, 2010.

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

[14] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[15] E D Dolan, J J More, and T S Munson. Benchmarking optimization software with cops 3.0. 5 2004.

[16] Andrew Fischer. Introducing circa: A dataflow-based language for live coding. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 5–8, 2013.

[17] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, page 947–954, New York, NY, USA, 2009. Association for Computing Machinery.

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

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Commun. ACM*, 62(12):56–65, November 2019.

[20] Mark Grechanik, Chen Fu, and Qing Xie. Automatically finding performance problems with feedback-directed learning software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 156–166. IEEE Press, 2012.

[21] Valentina Grigoreanu, Roland Fernandez, Kori Inkpen, and George Robertson. What designers want: Needs of interactive application designers. In *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, VLHCC '09, page 139–146, USA, 2009. IEEE Computer Society.

[22] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 837–847. IEEE Press, 2012.

[23] Alan C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, March 1993.

[24] David Kelk, Kevin Jalbert, and Jeremy S. Bradbury. Automatically repairing concurrency bugs with arc. In João M. Lourenço and Eitan Farchi, editors, *Multicore Software Engineering, Performance, and Tools*, pages 73–84, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[25] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: Insights from the practice. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1090–1101, 2018.

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

[27] 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 *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, 2012.

[28] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: automatically generating pathological inputs. pages 254–265, 07 2018.

[29] Remo Lemma and Michele Lanza. Co-evolution as the key for live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 9–10, 2013.

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

[31] Q. Luo, D. Poshyvanyk, A. Nair, and M. Grechanik. Forepost: A tool for detecting performance problems with feedback-driven learning software testing. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 593–596, May 2016.

[32] Ruchika Malhotra and Anuradha Chug. Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, 26:1221–1253, 10 2016.

[33] Brad Myers, Sun Young Park, Yoko Nakano, Greg Mueller, and Andrew Ko. How designers design and program interactive behaviors. In *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 177–184, 2008.

[34] Michael O'Neill. *Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution*. PhD thesis, University Of Limerick, Ireland, August 2001.

[35] S. Oney, B. Myers, and J. Brandt. Interstate: A language and environment for expressing interface behavior. pages 263–272, 2014.

[36] Stephen Oney, Brad A. Myers, and Joel Brandt. Euclase: A live development environment with constraints and fsms. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 15–18, 2013.

[37] Michael O'Neill and Lee Spector. Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*, 21, 06 2020.

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

[39] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. *CoRR*, abs/1708.08437, 2017.

[40] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 22–31, New York, NY, USA, 2014. Association for Computing Machinery.

[41] Afonso Jorge Ramos. Property Tests as Specifications Towards Better Code Completion. 2020.

[42] M. Riedl-Ehrenleitner, A. Demuth, and A. Egyed. Towards model-and-code consistency checking. pages 85–90, 2014.

[43] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

[44] Erik Sandewall. Programming in an interactive environment: The "lisp" experience. *ACM Comput. Surv.*, 10(1):35–71, March 1978.

[45] D.C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31, 2006.

[46] S. Sendall and W. Kozaczynski. Model transformation: the heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.

[47] Du Shen, Qi Luo, Denys Poshyvanyk, and Mark Grechanik. Automating performance bottleneck detection using search-based application profiling. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 270–281, New York, NY, USA, 2015. Association for Computing Machinery.

[48] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, page 532–543, New York, NY, USA, 2015. Association for Computing Machinery.

[49] Joanna Strug. Mutation testing approach to negative testing. *Journal of Engineering*, 2016:6589140, Jul 2016.

[50] Steven L. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

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

[52] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 314–326, New York, NY, USA, 2018. Association for Computing Machinery.

[53] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 364–374, USA, 2009. IEEE Computer Society.

[54] Chanakya Wijeratne and Rina Zazkis. On painter's paradox: Contextual and mathematical approaches to infinity. *International Journal of Research in Undergraduate Mathematics Education*, 1:163–186, 2015.

[55] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[56] Yongjie Zheng and Richard N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 628–638. IEEE Press, 2012.