FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Arbitraries with Confidence: Externalizing PBT inner-workings for better insights

Tiago Araújo Castro

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: André Monteiro de Oliveira Restivo

Second Supervisor: Hugo José Sereno Lopes Ferreira

July 19, 2021

Arbitraries with Confidence: Externalizing PBT inner-workings for better insights

Tiago Araújo Castro

Mestrado Integrado em Engenharia Informática e Computação

July 19, 2021

Abstract

Developers test what they expect, but bugs usually lurk in unexpected places. While unit testing consists of a system confirming a specific output given predetermined inputs, property-based testing also known as PBT combines the specification of properties with random input generation to test systems. Randomly sampling the possible inputs is not guaranteed to cover the whole input space, so, by relying on this technique, it must be accepted that the absence of evidence is not evidence of absence. PBT uses corner cases, which are less unexpected counterexamples, to conduct a biased sampling.

The developer is usually aware that 100% confidence concerning property satisfiability is generally unobtainable using these techniques, but they are also oblivious to a run's confidence. In essence, the question that remains is: "in the absence of a counterexample, what confidence does the developer have that it does not exist?". By providing detailed information about the input generation process, greater transparency is provided, resulting in the developer being able to make a deliberate choice to improve the tests or accept them as good enough.

A feature was developed complementing Fluent-Check, a PBT library written in TypeScript. The feature offers the developer the information needed to decide if they should increase their confidence in the tests, providing detailed information about the input generation process in various forms. Of all the PBT technologies analyzed, none contains a feature like the one proposed.

An empirical study was conducted with 16 participants, with results concluding that there is not enough statistical evidence to conclude that an informed testing process results in increased confidence in positive run results and control over input value distribution.

The contributions of this work include the implemented feature for the Fluent-Check library and the aforementioned empirical study.

Keywords: PBT, Statistics, Fuzzing

CCS concepts: CCS → Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Resumo

Os programadores testam o que esperam, mas os *bugs* geralmente escondem-se em lugares inesperados. Enquanto testes unitários consistem em confirmar um *output* específico dado um *input* predeterminado, a testagem baseada em propriedades, também conhecida como PBT, combina a especificação de propriedades com geração aleatória de *inputs* para testar sistemas. A amostragem aleatória dos *inputs* não garante a cobertura total do espaço possível de *inputs*, portanto, ao confiar nesta técnica, deve-se aceitar que a ausência de evidência não é evidência de ausência. PBT usa *corner cases*, que são contra exemplos menos inesperados, para executar uma amostragem tendenciosa.

O programador está normalmente ciente que 100% de confiança em relação à satisfatibilidade das propriedades geralmente não pode ser obtida usando estas técnicas, mas mantem-se abstraído da confiança de uma execução. Em essência, a questão que fica é: "na ausência de um contra exemplo, que confiança tem o programador de que ele não existe?". Ao fornecer informações detalhadas sobre o processo de geração de *inputs*, oferece-se maior transparência, fazendo com que o programador seja capaz de fazer uma escolha deliberada para melhorar os testes ou aceitá-los como bons o suficiente.

Uma funcionalidade foi desenvolvida para complementar o Fluent-Check, uma biblioteca de PBT escrita em TypeScript. A funcionalidade oferece ao programador as informações necessárias para decidir se este deve aumentar a sua confiança nos testes, fornecendo informações detalhadas sobre o processo de geração de *inputs* em várias formas. De todas as tecnologias PBT analisadas, nenhuma contém uma funcionalidade como a proposta.

Um estudo empírico foi realizado com 16 participantes, com resultados que indicam não haver evidência estatística suficiente para concluir que um processo de teste informado resulta num aumento da confiança no resultado positivo de uma execução e do controlo sobre a distribuição de valores de *input*.

As contribuições deste trabalho incluem a funcionalidade implementada para a biblioteca Fluent-Check e o estudo empírico mencionado anteriormente.

Palavras-chave: PBT, Estatística, Fuzzing

Categoria ACM: CCS → Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Acknowledgements

I want to thank my supervisors, André Restivo and Hugo Sereno Ferreira, for helping me become a better software developer. This was probably the project where I improved my coding skills the most, and I am thankful for that.

A huge thanks to my dad and my mom, Nuno, and Isabel, for all the love, for constantly pushing me to be a better person, and for providing me with a comfortable life. Lots of love to my grandfather, Manuel, for helping me understand the essential things in life and showing me that although life is hard, the moments you remember in your late years are the ones you cherish. I want to commend my younger brother, Rodrigo, for putting up with me all these years and take this opportunity to express my best wishes for his future. I also want to thank my cousin, Ricardo, for being like a brother to me all these years. To my godfather Carlos and my godmother Arminda I want to give my thanks for always being there for me.

I want to thank all of my friends, especially two groups, *Manolos* and *Roadinhos*, for being such a massive part of my life. I also want to give special thanks to my good friends João Africano and Pedro Martins for listening to my problems all these years and providing me with a safe space to share my thoughts and concerns.

To covid-19, it's about time you disappear. Stop being such a nuisance.

Lastly, I want to thank FEUP for being my second home for five years, even if the last two of those were mostly remote. I also want to thank all the professors that contributed to the knowledge I possess today and wish prosperity to this institution.

Tiago Castro

"The hardest choices require the strongest wills."

Thanos

Contents

List of Figures

List of Tables

Abbreviations

AFL	American Fuzzy Lop
CFG	Control-Flow Graph
PBT	Property-Based Testing
PRNG	Pseudorandom Number Generator
XOR	Exclusive OR

Chapter 1

Introduction

This chapter introduces the problem under analysis, going into detail about its context, motivation, objectives and presenting the document's structure. Section 1.1 describes the circumstances surrounding the work. Section 1.2 describes the importance of the problem being analyzed. Section 1.3 presents what is hoped to be demonstrated and what the reader can hope to learn from it. Section 1.4 presents the various sections of the document, objectively explaining each of them.

1.1 Context

Property-based testing [8], commonly referred to as PBT, combines the specification of properties with random input generation (fuzzing) to test systems as opposed to unit testing, which consists of a system confirming a specific output given predetermined inputs. A property is an aspect of a system that defines its mechanics together with other properties.

Properties have been present in people's minds since the earliest years of education. A good example are the properties of addition (1) Closure Property (in `add(a,b) == c` if type of *a* equals type of *b* then type of *c* must be the same), (2) Commutative Property (`add(a,b) == add(b,a)`), (3) Associative Property (`add(a,add(b,c)) == add(add(a,b),c)`), and (4) Additive Identity Property (`add(a,0) == a`).

To understand properties applied to technology, let us use an example. Imagine a group of developers who want to test an online store's ability to rate products from 0 to 5 stars, only taking whole stars (ratings can only be integers). To test this ability using unit tests, a large number of tests would have to be created:

- Assert that any random integer between 0 and 5 is valid;

- Assert that 0 and 5 are valid;
- Assert that any random value under 0 and another over 5 are invalid;
- Assert that any non-integer value (like a float) is invalid.

These tests not only take a long time to be created, but they are also not guaranteed to find any errors with the feature as these cases are relatively apparent. To test the feature using PBT, two properties can be created, (1) one testing the values that should be accepted (integers between 0 and 5, including both of them), and (2) other the values that should not be accepted (integers under 0 and over 5 as well as values of any other type such as float or string). This takes the responsibility of defining specific input values from the developer and results in a testing process that is less prone to human error.

1.2 Motivation

Because randomly sampling the possible inputs is not guaranteed to cover the whole input space, counterexamples may pass undetected. Some are less unexpected than others, and these are called corner cases. PBT uses them to do a biased sampling, implying some input values having a lower or higher sampling probability than others or being part of the sample by default.

Generally, 100% confidence concerning property satisfiability is unobtainable, so doubt will always remain in the developer's mind concerning a positive run's confidence. It is unlikely for a developer to feel secure about a test's positive result if the information on what was tested is not provided.

Let's suppose the developer creates a test that tests a system's ability to find a labyrinth exit. After a large number of random inputs, the program finds the exit. How confident can the developer be that there really is an exit? Does that confidence represent reality?

Let's now imagine that after the test, the developer can see the tested paths and notices that the program was generating paths going through walls. How confident can the developer now be that there really is an exit?

On top of all of this, there are many different PBT and fuzzing technologies, each claiming to have certain advantages compared to others. In many of these technologies, not much information is displayed to the developer about the process, limiting or even removing the ability to compare them.

1.3 Objectives

The primary objective is to extend the PBT library Fluent-Check [32] written in TypeScript to provide information for the developer so that they can assess the confidence of the executed tests. This poses several research problems, namely figuring out which information to display and how to display it. Of all the PBT technologies analyzed, none contains a feature like the one proposed.

It is expected that this feature will pave the way to improvements on the fuzzing engine, such as the use of gathered statistics for a more informed search.

1.4 Document Structure

This document is composed by five chapters, structured as following:

- Chapter 1 (p. 1), **Introduction** - introduces the problem under analysis, as well as its context, motivation, objectives, and the structure of the document;
- Chapter 2 (p. 5), **Background** - explores the key concepts surrounding the topics that are needed to understand this work including property specification, shrinking and PBT itself, featuring a brief explanation of its history;
- Chapter 3 (p. 13), **State of the Art** - describes the state of the art on the topics of search space optimization, metric visualization, and PBT;
- Chapter 4 (p. 25), **Problem Statement** - describes the problem in detail;
- Chapter 5 (p. 29), **Proposed Solution** - explains the created feature and goes into detail on how it works;
- Chapter 6 (p. 41), **Empirical evaluation** - describes the validation process;
- Chapter 7 (p. 57), **Conclusions** - summarises the developed work, presenting the expected results.

Chapter 2

Background

This chapter explores the key concepts surrounding the topics that are needed to understand this work. Section 2.1 introduces the main concepts of property-Based testing, referred to commonly as PBT, describing its origin, advantages, and drawbacks, diving into its history as well. The section also explains the concept of shrinking and property specification. Section 2.2 explains how Fluent-Check [32] works.

2.1 Property-Based testing

Property-based testing [8], commonly referred to as PBT, combines the specification of properties with random input generation (fuzzing) to test systems. This testing method differentiates from unit testing, which consists of a system confirming a specific output given predetermined inputs. This testing of user-defined outputs originating from user-defined inputs results in a bound to succeed test, not considering eventual counterexamples missed by the developer. Due to PBT using fuzzing, it can find these counterexamples, diminishing the room for human error. Section 2.1.1 contains some of PBT's history. Section 2.1.2 explains property specification and some methods for property creation. Section 2.1.3 explains what arbitraries are. Section 2.1.4 provides an explanation on how the shrinking process works.

2.1.1 History of PBT

PBT as a concept was introduced by Fink *et al.* [8], who presented its goals and the techniques needed to achieve them. While not defining all the aspects of the process, their idea of PBT used property specifications [7] and data-flow analysis of the program to guide the evaluation of test executions for correctness and completeness. The impossibility of representation with a finite set

of input data of the infinite number of possible computations is a raised concern, although not analyzed in depth.

Since then, PBT has risen as an alternative to the other testing methods. Various PBT tools have been created for multiple programming languages, the most famous one being QuickCheck [2], a free and open-source tool written in Haskell. The tool is analyzed in Section 3.1.1.

2.1.2 Property specification

Finding which properties to use can often be a challenge. When first engaging PBT, a common mistake developers make is defining redundant properties, which is expected since it is hard to know when the properties specified are enough to describe the system in question.

To show the importance of property selection, let's take addition as an example. Three of its properties are the commutative property (`add(a,b) == add(b,a)`), the associative property (`add(add(a,b),c) == add(a,add(b,c))`), and the closure property (in `add(a,b) = c` if type of "a" equals type of "b" then type of "c" must be the same) but these alone are not enough to represent addition. Other operations have these properties such as multiplication. If the additive identity property (`add(a,0) = a`) is added, the properties are no longer properties of multiplication. They are, however, properties of XOR. To differentiate addition from XOR, one must add another property stating: `sum(a,b) == sum(a-1,b+1)`. So, as shown, it is complicated to define something so correctly that its definition does not fit that of anything else.

Some approaches can help a developer specify properties. Scott Wlaschin [34] shared some of the strategies that help in this decision.

Different paths, same destination

Consists of combining operations in different orders to get the same example. An example presented by the author is the commutative property of addition;

There and back again

Consists of combining an operation with its inverse and ending up with the starting value. An example presented by the author is the serialization of a string to a binary format and its deserialization returning the original string;

Some things never change

Consists of the persistence of a particular aspect after executing a specific change. An example presented by the author is, after the sorting of a group of items, the same items still being present afterward;

The more things change, the more they stay the same

It is based on the concept of "idempotence" (property of a certain operation that can be applied multiple times without changing the result beyond the initial application). An example presented by the author is filtering the distinct types of elements in a set, always returning the same result after the first execution.

Solve a smaller problem first

It is based on "structural induction" (if a large object can be broken into smaller segments, and some property is true for these smaller segments, then you can often prove that the property is true for the large object as well). An example presented by the author is a property occurring in a sublist and inferring its presence in the whole list.

Hard to prove, easy to verify

It is based on the belief that an algorithm can sometimes be complex, but its verification reasonably simple. An example presented by the author is proving that a string tokenizer works by just concatenating the tokens together and getting the original string.

The test oracle

Consists of the testing of an algorithm using an alternate previously verified version. An example presented by the author is checking the result of a high-performance optimized algorithm by comparing its result to a much slower and easier to write algorithm.

2.1.3 Arbitraries

An arbitrary is a structure used to wrap information concerning a specific type, whether user-defined or built in the PBT technology. It usually contains a generator and a shrinking mechanism for the type in question. Most PBT technologies offer various built-in arbitraries (like integers and booleans) that allow developers to test multiple properties without specifying the used arbitraries.

There is often a need for the developers to use more complex input types in their properties, which is possible using built-in arbitraries but can result in undesired errors. The creation of user-defined arbitraries circumvents these problems. It is allowed by most PBT technologies providing the developer with the ability to control the generation of the data type and its shrinking method. User-defined arbitraries make use of built-in arbitraries for their definition.

2.1.3.1 Generators

In PBT, the generation of the data created to test the specified properties is done by generators. Although most PBT tools use random input generation, like observed in Section 3.1.2, some utilize different generation methods.

PBT tools sometimes use corner cases to increase the quality of the generation, improving the random generators. Corner cases are inputs that are more likely to cause a system to fail. PBT leverages this to do a biased sampling, implying some input values having a lower or higher sampling probability than others.

Arbitrary generators can use seeded or unseeded pseudorandom number generators, commonly called PRNGs, shared by the property's arbitraries. Seeded PRNGs allow the reproduction of executions, resulting in a better debug process by the developer.

2.1.4 Shrinking

Whenever a PBT run fails, the input that caused the failure is presented to the developer. However, a problem derived from random and other input generation types is the rather loose relation between the randomly chosen sample and the failing property's problem. That is why most PBT technologies contain a shrinking mechanism. Shrinking is a process that initiates after a counterexample has been found and tries to find a simpler one that also fails. Sometimes a shrunk counterexample may fail for a different reason, but that is rare and signifies deeper problems. The smallest the shrunk example is, the easier it is for the developer to recognize the property's real reason for failure. According to a repository by Link *et al.* [17] containing multiple shrinking challenges, the two major difficulties in shrinking are:

- Shrinking implies searching a large possibility space. Brute-forcing through all possible examples is usually not a very good option.
- When referring to the smallest example, the concept of "smallest" in a given domain or context is not always obvious.

Shrinking is a process that changes from technology to technology but always has the same goal despite the implementation. According to Hebert [11], the reduction of data through shrinking is generally made by transforming all the generators used and trying to bring them back towards their own zero point:

- A number tends to shrink from floating-point values towards integers, and integers tend to shrink towards the number 0;
- Binaries tend to shrink towards an empty binary («»);
- Lists also tend to shrink towards an empty list.
- Elements([A,B,C]) will shrink towards the value A

Hebert also states that larger recursive user-defined data structures may not be shrinkable, especially in statically-typed languages such as Haskell, where type declarations are used as generators instead of composable functions.

2.2 Fluent-Check

This section explains how Fluent-Check works and what functionalities it has to offer. Fluent-Check is a PBT library running on Node.js [9]. Section 2.2.1 explains the arbitraries used by the library. Section 2.2.2 explains how Fluent-Check approaches scenario creation.

2.2.1 Arbitraries

Fluent-Check contains the following default arbitraries (parameters containing a question mark are optional and "fc" refers to the Fluent-Check library):

Integer `fc.integer(min?, max?)`

Generates random integers from min to max. If min or max aren't specified, the minimum or maximum safe integers are used.

- Ex: `fc.integer(-500, 500)` - generates integers in [-500,500].

Real `fc.real(min?, max?)`

Generates random real values from min to max. If min or max aren't specified, the minimum or maximum safe integers are used.

- Ex: `fc.real(0, 1)` - generates real values in [0,1].

Boolean `fc.boolean()`

Generates a random boolean.

Array `fc.array(arbitrary, min?, max?)`

Generates random arrays with elements generated by a specified arbitrary. Min and max can be used to specify the minimum and maximum length of the arrays. If left unspecified, zero is used as the minimum length and ten as the maximum. Arrays of different lengths all have the same chance of being generated despite the number of inputs that can be generated with that length.

- Ex: `fc.array(fc.integer(), 0, 3)` - generates arrays from length zero to three containing integers from minimum to maximum safe integer.

Char `fc.char(start?, end?)`

Generates a random character from 'start' to 'end' using ASCII values. Both parameters signify a range of values from the ASCII table which can be generated (only generates valid characters). If only 'start' is specified, only that character is generated. If none are specified, every usable character can be generated.

- Ex: `fc.char('a', 'z')` - generates lower case alphabetical characters

String `fc.string(min?, max?, char?)`

Generates random strings with length from min to max. If left unspecified, min is assigned a value of two, and max a value of ten. To specify the type of characters the string contains, a char arbitrary can be determined (if not specified, it generates all types of characters). Strings of different lengths all have the same chance of being generated despite the number of inputs that can be generated with that length.

- Ex: `fc.string(0, 8, fc.char('a', 'Z'))` - generates strings from length zero to eight containing alphabetical characters

Set `fc.set(elements, min?, max?)`

Generates all combinations of the given elements with lengths from min to max. If min or max are not specified, then min is assigned the value of zero, and max becomes the number of elements (if the number of elements is higher than ten, max is assigned the value of ten).

- Ex: `fc.set(['a', 'b', 'c', 'd'])` - generates all combinations of the characters "a", "b", "c" and "d" from length zero to four.

Union `fc.union(...arbitraries)`

Receives various arbitraries as a parameter and randomly chooses one of them to create the input (arbitraries with a higher number of possible inputs have a higher chance of being generated).

- Ex: `union(integer(0,10), integer(70,100))` - generates integers from zero to ten or integers from seventy to one hundred.

Tuple `fc.tuple(...arbitraries)`

Receives various arbitraries as a parameter and generates an array containing a random value of each arbitrary, creating multiple inputs in a single one.

- Ex: `fc.tuple(fc.integer(-10,10), fc.boolean())` - generates an array containing a random integer from negative ten to ten and a boolean (ex: [-5, false]).

Other arbitraries can be created using the "chain", "filter" and "map" arbitrary methods.

2.2.2 Scenario creation

Fluent-Check uses a fluent interface where changes to the scenario are done via methods called from it, returning a scenario class representative of the method used containing a reference to the class from which it was called (parent).

A scenario is used to configure a property's assertion and input generation procedure. To create a new scenario, "`fc.scenario()`" is used ("fc" refers to the fluent-Check library). To specify a scenario's arbitraries and configure it, the following methods are used:

- `config(strategy)` - can be used to configure multiple aspects of the scenario;
- `forall(name, arbitrary)` - states that for all values generated by this arbitrary, the property must hold (∀ - universal quantification);
- `exists(name, arbitrary)` - states that for at least one value generated by this arbitrary, the property must hold (∃ - existential quantification);

- `given(name, function)` - creates a variable which is defined by the function's return. The function can receive the run value associated to whichever variable was defined previously using "forall", "exists" or "given".
- `when(function)` - used to make changes on the generated values via a function passed as parameter.
- `then(assertionFunction)` - receives a function that receives as parameters the values generated by the previously set arbitraries and returns a boolean;
- `and(function)` - can only be used after "given", "when" and "assert" and works the same as the preceding call.
- `check()` - used to run the scenario.

For each arbitrary associated with the scenario, it will generate 100 values (can be changed using the "config" method). Since every combination of the generated inputs is provided to the assertion, it will be tested 100^(number of arbitraries) times.

Chapter 3

State of the Art

This chapter describes the state of the art on the topic of property-based testing, also known as PBT. Section 3.1 analyzes the existing technologies concerning PBT. Section 3.2 tackles the existing technologies concerning fuzzing. Section 3.3 explains how some technologies contain features closely related to the one proposed by this work. Section 3.4 contains a summary of what was learned in this chapter.

3.1 PBT technologies

This section describes some of the PBT technologies that exist. The technologies analyzed are the ones that showed more use from the community and seemed more relevant to the topic of this work. Section 3.1.1 contains an analysis of the various technologies and, if existent, the respective papers. Section 3.1.2 gathers and compares the input generation methods of all the analyzed technologies. Section 3.1.3 gathers and compares the transparency provided by the technologies. Section 3.1.4 analyzes some of the approaches that PBT technologies use for shrinking.

3.1.1 Analysis

QuickCheck is a Haskell tool developed by Claessen *et al.* [2] for property testing and formulation, presenting and analyzing successful case studies of its use. The tool is referenced by many other PBT tools and provides (1) type-based default random input test data generators and (2) an embedded language for specifying custom test data generators. The authors state that random testing is most effective when the distribution of test data follows that of actual data. Still, in most

cases, this is not possible, so a uniform distribution is often used. However, it is not feasible for data drawn from infinite sets, so distribution is put under the human tester's control. The choice for the use of random input generation is justified this way: (1) for small programs, in particular, random test cases will likely cover all paths, (2) many criteria needing reinterpretation if systematic methods were used, resulting in heavier methods, and (3) the belief that there is no reason to believe that systematic methods would yield better results. After a failed execution, QuickCheck provides the number of tests after which each property failed, the number of shrinking steps, and the counterexample found. After a successful execution, only the number of passed tests are presented. This tool uses type-based shrinking, which is defined based on the value's type. Every value of the same type shrinks the same way, regardless of how it was generated.

Another famous PBT software is **fast-check**, a framework developed by Dubien *et al*. [6] for JavaScript and TypeScript. The framework's repository showcases the reasons why a developer should use fast-check by demonstrating its advantages and demonstrates some of the issues found in famous packages by using it. The author explains many of the framework's features, including a simple and straightforward explanation of runners and arbitraries. Arbitraries are responsible for the random but deterministic generation of values, and runners are responsible for running, executing, and checking that properties stay true whatever the generated value. After a failed execution, fast-check provides the number of tests after which each property failed, the seed of the run for further replication, a counterexample, the number of shrinking steps, and encountered failures. After a successful execution, no extra data is provided except for the tests' execution time. Fast-check uses a type-based shrinking strategy where every value of the same type shrinks the same way.

Created for Java, **JQF** is an open-source tool for performing coverage-guided fuzz testing created by Padhye *et al*. [27, 26]. JQF is built on top of **junit-quickcheck** developed by Holser *et al*. [13] and has been successful in discovering 42 previously unknown bugs in open-source software. The way the coverage guidance works is by continuously executing the program with randomly generated inputs. If a new input leads to an increase in code coverage, it is saved for subsequent mutation. The tool distinguishes itself from other tools because it features various types of guidance (1) using Zest [28] algorithm specifically designed for coverage-guided property testing, (2) using the AFL [36] tool, (3) following the PerfFuzz [16] technique for automatically generating test inputs that maximize performance counters, such as loop execution counts and (4) Repro Guidance which is not analyzed in depth by the author. The tool also allows no guidance, which results in a random generation just like junit-quickcheck. After consulting the current capabilities of the tool, Repro Guidance is no longer available, and two new types of guidance were added, one using RLCheck [30], based on reinforcement learning, and another using BigFuzz [37], a coverage-guided fuzz testing tool for big data analytics. Junit-quickcheck's output on a fail consists of the original failure message and arguments, the shrunken argument, and the execution's seed so the developer can repeat the run. JQF's output depends on the guidance chosen. Informa-

tion on some of the various guidances used can be found on Section 3.2. In JQF, shrinking works as in junit-quickcheck and is integrated into the generation process. The generator is in charge of controlling how the values it produced are shrunk.

Reporting on the design and implementation of, to their knowledge, the first property-based framework for quantum programs, Honarvar *et al.* [14] introduce **QSharpCheck**, a property-based testing framework written in Q# and apply the tool to two case studies, measuring their approach's effectiveness using a mutation score. In this tool, which uses random input test data, the specified properties receive as one of the parameters a target confidence level for accepting or rejecting the statistical hypothesis. Using the statistical method of hypothesis testing, while assuming a binomial distribution of measurement probabilities, QSharpCheck considers the assertion provided in the last part of the test to be the null hypothesis based on the confidence level provided as the initial parameter and the measurement performed for each generated test case. After an execution, QSharpCheck presents the previously set confidence level, number of test cases, number of measurements, number of experiments, and the test execution time. When the execution fails, it also outputs θ and ϕ relative to the counterexample qubit value, which is specific to quantum computing. Figure 3.1 shows the example of a test property using QSharpCheck. The first line is the property's name. The next one refers to the parameters, which are all optional (number of test cases, confidence level, number of measurements, and number of experiments per concrete case, respectively). The third line concerns the input data's allocation and setup (in this case, a single qubit). The last two lines contain the system under test ("TransformState") and the test code, which in this case is a built-in assertion method of the library. QSharpCheck does not feature shrinking, a feature common in PBT technologies.

```
1 Transform_Property;
2 (10, 99, 500, 300);
3
4 (q : Qubit (36,72)(0,360));
5 TransformState(q);
6 [AssertTransformed(q,(108,144)(0,360))];
```

Figure 3.1: QSharpCheck test property specification (fetched from [14])

Hypothesis created by MacIver *et al.* [19] is an easily extendable property-based testing library for Python that has various extensions such as hypothesis-networkx6 and hypothesis-bio7. The authors mention that traditionally PBT consists of random test-case generation, followed by test-case reduction (shrinking), requiring the users to manually specify a validity oracle (predicate identifying an arbitrary test case validity). The limitations of this are specified as being that (1) to improve on random generation's ability to find bugs, it will normally require a modification of the tests to support new ways of generating data which are usually more complex, and that (2) users will rarely be willing to undertake this work themselves, making researchers work more

per project to understand how to test it. Hypothesis avoids both of these problems by using a single universal representation for test cases. The library supports Targeted Property-Based Testing [18], which uses a score to guide testing towards a particular goal specified by the user. By guiding the input generation towards values with a higher probability of falsifying a property, the input space is searched more effectively, requiring fewer tests to find a bug or achieve high confidence in the tested system than random PBT. After a failed execution, hypothesis presents the user with the counterexample, not providing any more information. After a successful execution, no extra data is provided except for the execution time. In this tool, shrinking is integrated into the generation process, and the generator controls how the values produced by it shrink.

Papadakis *et al*. [29] present **PropER**, a property-based testing tool integrated with the Erlang type language. PropEr uses random input generation, uses types as generators automatically, and turns function specifications into simple properties. A function signature, also called spec, is a form of lightweight specification that defines the function's input and output. The tool contains a method for testing functions automatically by testing it against its spec, calling it with increasingly complex valid inputs, and checking that no unexpected value is returned, allowing for the creation of simple testable properties. When shrinking recursive types such as lists, instead of generating the sub-instances as the tuple is built, PropER generates and then combines them into a tuple improving the process's effectiveness. The use of types as generators reduces code redundancy as the type declaration becomes the single point of specification for data types. On a failed run, PropER's output consists of a counterexample presented and then shrunk, also displaying the number of times it was shrunk. On a successful run, it simply outputs the number of passed tests and nothing else. In this tool, shrinking is integrated into the generation process, and the generator controls how the values produced by it shrink. This process can often be fine-tuned through user-defined shrinking strategies.

Löscher *et al*. [18] present targeted property-based testing, which aims to make the input generation component of a property-based testing tool guided by a search strategy. **Target** is an integrated component of PropEr [29] that is also an Erlang framework that combines the advantages of both search-based and property-based testing. It is used when the properties to be checked have a form that involves a utility value to maximize or minimize. It consists of three main components: (1) the strategy that is used to explore the input space, (2) the component that supports writing targeted generators, and (3) utility values, which are paired with the input to the property. Target contains two search strategies, (1) Hill Climbing and (2) Simulated Annealing. The former is an iterative algorithm that starts with an arbitrary solution to a problem, attempting to find a better one by changing it. If the change results in an improved solution, another change is made until an optimum is found. The latter works the same way but allows downward steps in order to escape from a local optimum. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited. The authors present 3 test cases where random input generation is compared to Target, and in all of them, a superiority in the use

of target is shown. Target's output is the same as PropER's. On a failed run, the output consists of a counterexample presented and then shrunk, also displaying the number of times it was shrunk. On a successful run, it simply outputs the number of passed tests. Target's shrinking process is also the same as PropER's.

3.1.2 Input generation

As seen in the previous subsection, many PBT technologies use different input generation methods. The study of the different methods used allows for a better comprehension of search space optimization. In Table 3.1, the various PBT technologies are associated with the respectively used input generation methods. The various input generation methods specified in the table are respectively:

- **Random** input generation is a black-box software testing technique where programs are tested by generating random, independent inputs;
- **Targeted** PBT guides the input generation towards the maximization or minimization of a utility value through the use of methods such as Hill Climbing or Simulated Annealing;
- **Binary** fuzzing consists of finding a simple binary and executing the program over and over again with slightly different, randomly mutated versions of the binary in each run;
- **Semantic** fuzzing guides the input generation towards a better search of the semantic analysis stage of programs, maximizing code coverage;
- **Complexity** fuzzing generates inputs that exhibit worst-case algorithmic complexity in different components of the program.

Table 3.1: Input generation methods used by each PBT technology

Technology	Random	Targeted	Binary	Semantic	Complexity
QuickCheck [2]	•				
fast-check [6]	•				
QSharpCheck [14]	•				
JQF [27]	•		•	•	•
junit-quickcheck [13]	•				
Hypothesis [19]	•	•			
PropER [29] + Target [18]	•	•			

3.1.3 Transparency

As seen in the analysis of the various PBT technologies, many of them provide different information on the run. In Table 3.2, the various PBT technologies are associated with the respectively provided information on a successful run. The information presented on a failed run, although not as relevant to this work's objective, which is to provide confidence to the developer relative to the evaluated test's satisfiability, is presented in Table 3.3.

Table 3.2: Information provided by each PBT technology after a successful execution

Technology	Number of test cases generated	Execution Time
QuickCheck [2]	•	
fast-check [6]		•
QSharpCheck [14]	•	•
junit-quickcheck [13]	•	
Hypothesis [19]		•
PropER [29] + Target [18]	•	

JQF isn't included in this list, because the information presented depends on the chosen fuzzing technology. All of them can be consulted on Section 3.2.

It is important to note that the confidence level provided by QSharpCheck as output is previously set by the developer, resulting in it not contributing to transparency but instead to the configurability of the technology.

Table 3.3: Information provided by each PBT technology after a failed execution

Technology	Counterexample	Nº of tested cases	Nº of shrinks	Seed	Execution time
QuickCheck [2]	•	•	•		
fast-check [6]	•	•	•	•	
QSharpCheck [14]	•	•			•
junit-quickcheck [13]	•			•	
Hypothesis [19]	•				
PropER [29] + Target [18]	•		•		

JQF isn't included in this list, because the information presented depends on the chosen fuzzing technology. All of them can be consulted on Section 3.2.

3.1.4 Shrinking

There are various ways to approach shrinking implemented by PBT technologies. According to two articles by MacIver [21, 20], the creator of Hypothesis, on the library's website, shrinking can be:

- **Type-based** - The shrinking process is based on types. Any value of a given type shrinks the same way, regardless of the generation, saving the developer some time, but can originate worst results.

- **Integrated** - The shrinking process is defined when specifying a generator, allowing it to control the shrinking process. It requires a higher amount of effort but allows for domain-specific, targeted shrinking.
- **Compositional** - The shrinking process works by taking the random bitstream used to generate the argument instance and re-running the property test while simplifying the bitstream, simplifying the counterexample in turn.

Some PBT technologies such as QuickCheck [2] do the first, while other technologies such as Hypothesis [19] do the second. Compositional shrinking is not as common as the other two and is implemented by technologies such as theft developed by Vokes *et al*. [33] and QuickTheories developed by Coles *et al*. [3]. Integrating shrinking into the generation has two large benefits compared to Type-based shrinking:

1. Provides the ability to shrink anything the developer wants to generate regardless of whether there is a defined shrinker for the type produced.
2. The developer can guarantee that the shrinking process upholds the same invariants as the generation process. This is especially important because, without the same invariants being satisfied by both the shrinker and the generator, important errors may be reduced to completely different errors.

In Table 3.4, the analyzed PBT technologies are associated with their shrinking behaviors.

Table 3.4: Shrinking behaviours used by each PBT technology

Technology	Type-based	Integrated	Compositional
QuickCheck [2]	•		
fast-check [6]	•		
QSharpCheck [14][1]			
JQF [27] + junit-quickcheck [13]		•	
Hypothesis [19]		•	
PropER [29] + Target [18]		•	
Theft [33][2]			•
QuickTheories [3][2]			•

[1] QSharpCheck does not feature shrinking.
[2] These technologies were presented by MacIver [20] as examples of technologies that use a compositional shrinking methodology

3.2 Fuzzing technologies

AFL is a security-oriented binary fuzzer created by Zalewski *et al*. [36] and defined as a collection of hacks that have been tested in practice and found to be surprisingly effective. The fuzzer

combines multiple deterministic and non-deterministic fuzzing strategies with genetic algorithms to increase code coverage and has no domain-specific knowledge of input structure as it simply views inputs as sequences of bytes. AFL uses binary fuzzing, which consists of finding a simple binary and executing it repeatedly with slightly different, randomly mutated inputs in each run. It presents a real-time status screen visible in Figure 3.2 containing various types of information concerning the fuzzing process using colors to keep data readable and attract the developer's attention to the most important details. The screen is divided into multiple sections, each containing different information. The most relevant being:

- **Process timing** - contains information on how long the fuzzer has been running and how much time has elapsed since its most recent finds;
- **Overall results** - contains information on (1) the times the fuzzer went over all the interesting test cases discovered, (2) the number of test cases discovered, and (3) the number of unique faults;
- **Cycle progress** - contains information on how far along the fuzzer is with the current queue cycle;
- **Map coverage** - provides some trivia about the coverage observed;
- **Path geometry** - tracks the path depth that was reached through the action of the guided fuzzing process;
- **CPU load** - displays the CPU utilization.

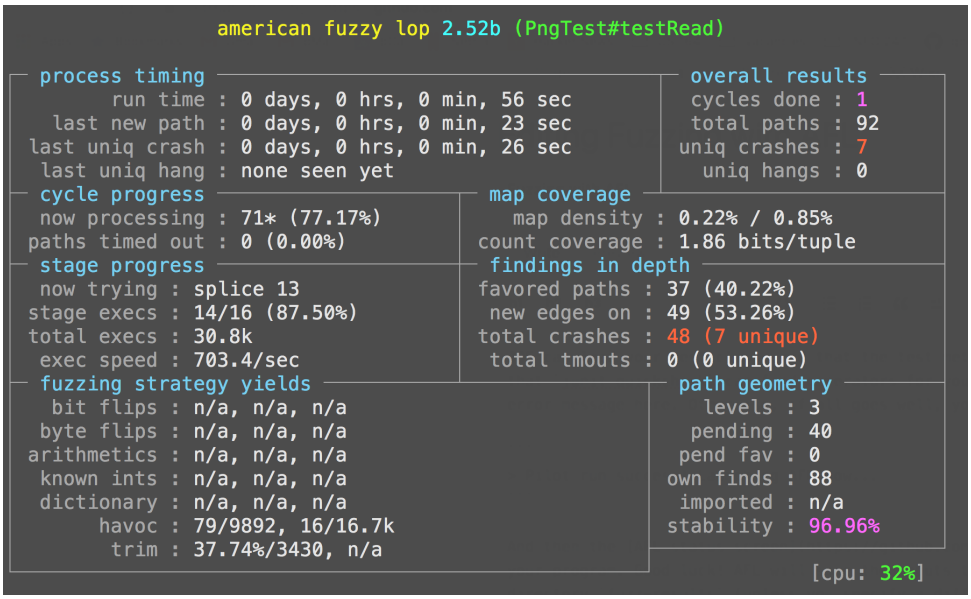


Figure 3.2: AFL status screen (fetched from [26])

Padhye *et al*. [28] present **Zest**, a semantic fuzzing technique implemented on top of JQF [27], which guides random input generators for a better search of the semantic analysis, maximizing code coverage. It incorporates ideas from coverage-guided fuzzing into generator-based testing and successfully tests programs' semantic analysis stage, complementing tools such as AFL [36]. Zest first converts a random input generator into a parametric generator, which is a function that takes a sequence of untyped bits and produces a structured input as opposed to each input being decided by a pseudo-random source of bits. After this, Zest guides parametric generators to produce inputs that get deeper into the semantic analysis stage of programs using feedback-directed parameter search, which works by analyzing inputs, and if they are valid and contribute to better code coverage, the parameter sequence is added to the list of important test inputs, which are used as candidates for future mutations ensuring that Zest keeps mutating valid inputs that exercise core program functionality. Zest features a status screen visible in Figure 3.3 that displays various information on the method being fuzzed and various statistics about the fuzzing process such as (1) number of executions, (2) valid inputs, (3) completed cycles, (4) unique failures, (5) execution speed, (6) total and valid coverage, among others. Total and valid coverage refers to the branches that have been exercised by all inputs and by valid inputs.

```
Zest: Validity Fuzzing with Parametric Generators
-------------------------------------------------
Test name:             CalendarTest#testCompare
Results directory:     /path/to/tutorial/fuzz-results
Elapsed time:          15s (no time limit)
Number of executions: 6,449
Valid inputs:          6,368 (98.74%)
Cycles completed:      1
Unique failures:       1
Queue size:            26 (5 favored last cycle)
Current parent input: 16 (favored) {272/920 mutations}
Execution speed:       480/sec now | 421/sec overall
Total coverage:        14 (0.02% of map)
Valid coverage:        14 (0.02% of map)
```

Figure 3.3: Zest status screen (fetched from [26])

Defined as a method to automatically generate pathological inputs without any domain knowledge about the program, **PerfFuzz** developed by Lemieux *et al*. [16] implements complexity fuzzing via feedback-directed mutational fuzzing, and independent maximization of the execution counts for all program locations allowing itself to find various inputs that generate others with higher total execution path length than previous approaches. Pathological inputs are those inputs

that maximize the execution count of a particular program component given a fixed input length. PerfFuzz functions in a similar way to Zest [28]. It uses seed inputs designed to verify the program's functional correctness, and in the absence of such seeds, arbitrary inputs may be used. The feedback-directed fuzzing aims to maximize each edge's execution count in the control-flow graph (CFG) of a program, adding mutated inputs that do this to the important test inputs list used as candidates for future mutations. PerfFuzz is built on top of AFL [36], so it has no domain-specific knowledge of input structure as it simply views inputs as sequences of bytes. PerfFuzz saves mutated inputs if they maximize the execution count for any CFG edge, even if the mutation reduces the total execution path length, which allows the finding of better results. The process' output is the top three CFG edges for each of the three favored inputs, represented by start and end line numbers and by their execution count, as shown in Figure 3.4.

Input #9189		Input #10520		Input #10944	
Exec. count	CFG edge	Exec. count	CFG edge	Exec. count	CFG edge
2,071,824	pngrutil.c:3715->3715	289,536	pngrutil.c:3842->3842	225,489	pngread.c:387->396
274,212	pngrutil.c:3715->3712	144,536	pngrutil.c:3416->3419	225,489	pngread.c:405->456
274,178	pngrutil.c:3712->3715	144,536	pngrutil.c:3419->3404	225,489	pngread.c:456->459

Figure 3.4: PerfFuzz example output (fetched from [16])

3.3 Related work

AFL [36] and Zest [28] are the technologies that come the closest to what this work sets out to achieve. Both of them provide real-time information that contributes to a better understanding of the run's satisfiability, increasing the whole process's transparency. According to this work's objective, the most crucial statistic presented by them is code coverage. AFL displays the percentage of branch tuples visited for the current input and all the inputs. Zest provides information on the number of branches that have been exercised by all inputs and by valid inputs.

3.4 Summary

After analyzing various PBT and fuzzing technologies, AFL [36] and Zest [28] demonstrated some features desired by this work and QSharpCheck [14] demonstrated how confidence quantification could be a useful feature, although this work does not set out to achieve that. However, all the other technologies showed a lack of transparency associated with their process.

All PBT technologies except for JQF [27], upon a successful run, showed mostly only previously user-set data such as the number of tested cases, which is mainly limited by the developer, and the test's execution time. This amount and type of information does not improve the run's transparency, failing to provide the developer with a high enough confidence in the result shown. JQF, by featuring Zest and AFL, provides some vital information to the developer. However, it

fails to address other vital metrics such as generated input distribution which can be represented in various ways such as graphs.

Chapter 4

Problem Statement

This chapter describes the problem and the proposed solution to solving it, detailing the preliminary work, and defining the work plan. Section 4.1 explains the problem under study, stating the difficulties associated. Section 4.2 states the research questions. Section 4.3 explains the validation methodology. Section 4.4 provides a summary of everything discussed in this chapter.

4.1 Problem under study

Generally, 100% confidence concerning property satisfiability is unobtainable, so doubt will always remain in the developer's mind concerning a positive run's confidence. It is unlikely for a developer to feel secure about a test's positive result if the information on what was tested is not provided.

As concluded by the research done in Chapter 3 (p. 13), most PBT technologies fail to provide a high enough level of transparency associated with their runs. Only JQF [27], via its integration of AFL [36] and Zest [28], does something similar to what is intended by this work.

The primary objective is to extend the PBT library Fluent-Check [32] written in TypeScript to provide information for the developer so that they can assess the confidence of the executed tests. This poses several research problems, namely figuring out which information to display and how to display it. Of all the PBT technologies analyzed, none contains a feature like the one proposed.

The main proposed features to include in the proposed feature are:

- Presenting the generated input values in an organized fashion containing information related to their execution;

- Creating graphs representing the distribution of the input values;
- Providing the seed of the execution for repeatability purposes.

Other possible features include: (1) presenting the time taken for a property to be evaluated and (2) representation of the input space coverage.

4.2 Research questions

When trying to understand the impact of a feature like the one proposed, it is expected that, by using it, developers will obtain better insight into the generation process and consequently, try to control and change it to match their intentions, increasing the confidence they have in the quality of their tests.

As such, the following hypothesis is presented:

Hypothesis: An informed testing process in PBT leads to higher control over input value distribution and higher confidence in positive run results.

An informed testing process refers to the proposed feature. Various things contribute to higher transparency. The proposed solution exposes a few of the metrics that achieve this, such as a visual representation of the generated inputs' distribution and creating a file containing all the values generated for the execution in question. The control over input value distribution is measured by the number of changes a developer makes to its generation process and their ability to know what generation they should use in each property. Confidence in positive run results refers to the degree of belief that a developer has that the positive result that a test expressed is correct. A positive result is obtained by an execution finding no input values that break the properties.

The following research questions will also be answered in this dissertation to grasp the effects of the proposed feature.

RQ1: *Does an informed testing process increase confidence in the positive result of a run?*

This research question aims to ascertain whether the metrics presented to the developer result in them expressing higher confidence in a positive result. The data used to answer this research question is subjective since it requires developers to express their opinions.

RQ2: *Does an informed testing process increase insight and control over the generation of input values?*

This research question aims to ascertain whether the metrics presented to the developer result in a higher capability from them to control the generation process and bend it to their will. The data used to answer this research question is objective and consists of the number of changes the participants make after first defining a property's generation.

4.3 Validation

An empirical test is to be performed to validate the proposed hypothesis and answer the research questions, in which two groups of programmers execute a pair of tasks where they are expected to debug and improve the provided code. One group has access to the developed feature and the other only to the basic Fluent-Check.

Task 1

This task provides the developer with a function and some properties testing it. The participants are told to find any bugs in the code while increasing the properties' quality and creating any properties they might find necessary to define the system in question.

Task 2

This task presents to the developer a more complex function. The participants are expected to find any bugs in the code while creating the properties they believe are necessary to define the system in question.

The time taken to solve each task, information on how participants interact with the proposed feature and Fluent-Check itself, and the final code are all recorded, providing the data points to answer the research questions of this dissertation.

A thorough account of the Validation process, including the methodology and a discussion of the results obtained, can be found in Chapter 6 (p. 41).

4.4 Summary

As concluded by the research done in Chapter 3 (p. 13), most PBT technologies fail to provide a high enough transparency level associated with their runs. This work aims to increase Fluent-Check's processes' transparency, consequently providing the developers with a higher level of confidence relative to the run's result.

As part of this work, a feature was envisioned which presents the developer with multiple metrics and statistics in various ways. Furthermore, the feature's validation is to be made by conducting empirical tests, comparing various participants' testing processes, allowing the research questions to be answered and the hypothesis validated.

Chapter 5

Proposed Solution

This chapter presents a description of the implementation, providing an overview in Section 5.1 and a more detailed approach explaining each feature's objective and implementation in Section 5.2. A small summary of everything mentioned in this chapter is also present in Section 5.3.

5.1 Overview

The collective objective of the various features is the enhancement of the developer's insight into the input generation process. In this section, the various features implemented are described succinctly.

The reporter feature handles all console output and file generation while all the others focus on providing the reporter with the needed information. The seeded generator feature increases not only the transparency of the library but also its configurability. The execution time feature allows the developer to see how changes to the code reflect in its efficacy. The test case output feature provides the developer with a configurable CSV file containing all the test cases generated, the run result, and their execution time so strange behaviors can be more easily detected. The input space coverage feature provides the developer with a percentile value resembling the amount of input space covered so the developer can better understand how to control the generation. Finally, the graph creation feature allows for a better view of test case distribution in the input space by generating an SVG file resulting in a better understanding of the input generation process.

Table 5.1 (p. 30) provides an overview of which information is presented to the developer and how.

Table 5.1: Feature output overview

Feature	Console output	CSV file	SVG file
Seed	•		
Execution time	•		
Test cases		•	
Input space coverage	•		
Graphs			•

5.2 Features

Fluent-Check [32] is a PBT library running on Node.js [9] that uses a fluent interface where changes to the scenario are done via methods called from it, returning a scenario class representative of the method used containing a reference to the class from which it was called (parent). In order to configure which information is presented in the end of a run, the developer should use the scenario method: `configStatistics( fc.statistics())` ("fc" refers to the Fluent-Check library). This method receives a statistics object which also uses a chain architecture. The various pieces of information that can be presented at the end of a property execution are presented in this section along with the statistics object method used to enable each of them (except for the seeded generator, which is specified from the scenario, not from the statistics object).

5.2.1 Reporter

Because the proposed functionality is all about providing the developer with information about the input generation process, a way to deliver that information is necessary, so the reporter function was created. This function can be called using `fc.expect(scenario)` and receives as a parameter a scenario's execution. It takes the object returned by the scenario containing all the information provided by the other features and takes care of all the information supplying. Most of the information provided by the features requires a simple print to the console. However, the reporter function is also in charge of generating the CSV files containing all the test cases and associated information, explained in detail in Section 5.2.4, and graph creation using Data-Driven Documents, also known as D3 [1], explained in detail in Section 5.2.6.

An example of the console output of the reporter function when a property is deemed satisfiable is present in Figure 5.1 (p. 31). In this case, the information is written to the console using `console.log(information)`. To enable the printing of information when a property is satisfiable, the `fc.statistics()` object method `withOutputOnSuccess()` must be used (or `withAll(csvPath?, csvFilter?)` which encompasses this functionality as well as the test case output functionality explained in Section 5.2.4 and the input space coverage functionality

explained in Section 5.2.5). The scenario used for this example states: "For all integers ('a') from zero to one hundred and all arrays ('b') from zero to three length containing integers from negative five to five should follow a particular assertion.". The scenario enables all of the features.

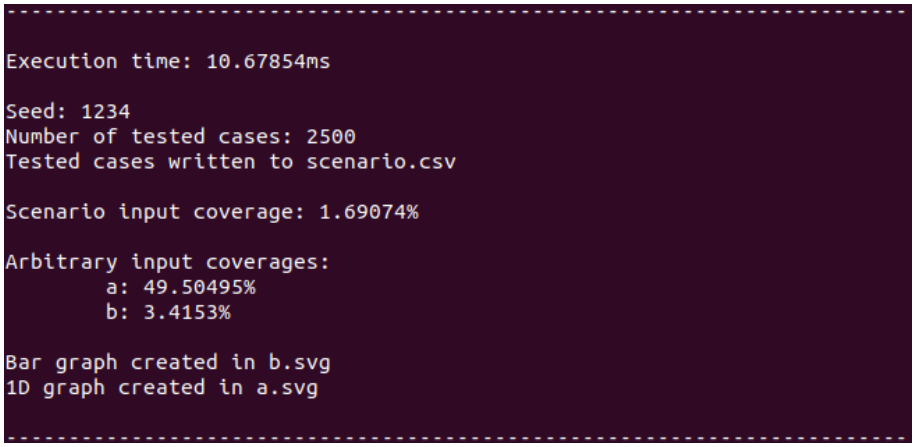


Figure 5.1: Example of a reporter function output when the property is satisfiable

An example of the console output of the reporter function when a property is deemed unsatisfiable is also present in Figure 5.2. In this case, the information is written to the console by throwing an error containing it as the message, later to be handled by the used test framework, which in this case is mocha [12]. The scenario states the same as the scenario used to exemplify the reporter's output on a successful test but is forced to fail whenever seventy-five or seventy-six is generated for 'a'. In this output, there are two visible differences: (1) the color of the text due to the way the information is written to the console, and (2) the counterexample found.

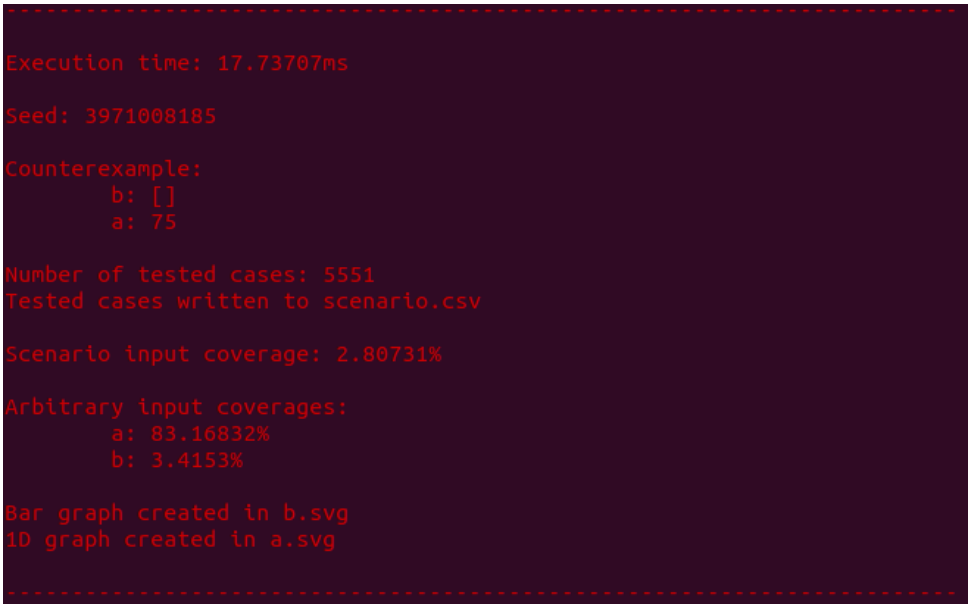


Figure 5.2: Example of a reporter function output when the property is unsatisfiable

5.2.2 Seeded generator

The default pseudorandom number generator, also known as PRNG, used by Fluent-Check is `Math.random`, which not only does not provide the seed used for the number generation but also does not allow a seed to be specified when it is called. This was an issue because it did not allow developers to reproduce previous executions. The solution found was to allow them to specify a PRNG to be used by the scenario in charge of the property configuration and execution. The PRNG specified by the developer should return a real number between zero and one.

In order to specify the desired PRNG and the seed to be used (if not specified, a random one is generated), the developer should use the scenario method `withGenerator(generator, seed?)`. The generator function must have the following signature: `(seed: number) => () => number`. This signature makes it so the generator can be instantiated only when a scenario is first executed, and pseudorandom numbers can be fetched from the generator by simply calling the instance. Instantiating the generator on the first execution allows further reruns of the executions with the same seed to produce the same test cases.

The seed used in the executions is presented at the end of the property run, provided by the reporter, which is explained in Section 5.2.1.

An example of a possible PRNG function is: `(seed: number) => () => (seed = seed * 16807 % 2147483647) / 2147483647`

5.2.3 Execution time

When testing code in a large project, developers can usually end up with a large folder for their tests. Optimization of these tests is essential to save time, and an excellent way to measure the impact of various code changes is by comparing the execution times of various test runs.

The execution time is calculated using performance-now [23] which is a function similar to `performance.now` that is present in modern browsers and gives time in milliseconds but with sub-millisecond precision. Performance-now offers the same function based on the Node.js native function: `process.hrtime`. Using this native function means that the reported time will be monotonically increasing and not subject to clock drift which refers to several related phenomena where two clocks do not run simultaneously.

The execution time is displayed in milliseconds and is shortened to five decimal places.

5.2.4 Test case output

When testing a property's satisfiability, many test cases are generated. This feature's objective is to offer the developer the ability to generate CSV files using the test cases and each of their results and execution times.

To enable the file's creation, the method `withTestCaseOutput(path?, filter?)` called from the object returned by `fc.statistics()` must be used (or `withAll(csvPath?, csvFilter?)`

which encompasses this functionality as well as the output of information when the analyzed property is deemed satisfiable explained in Section 5.2.1 and the input space coverage functionality explained in Section 5.2.5).

As for the available parameters, the first is the path where to write the file, which must contain the file's name in the end without the extension and, if left unspecified, writes it to the current directory. The second parameter is the filter function which is applied to each generated input and should follow the following signature: `(testCase: Object, execTime: number, result: boolean) => Object | undefined`. It receives as parameters an object containing the values generated for each arbitrary, the run's execution time, and the result of the run. The filter should return an object containing keys referring to the file column's names and their assigned values for each row. In order to ignore specific executions, an undefined value can be returned, and that row will not be written to the file.

Table 5.2 shows the first ten lines of the default file generated by the same scenario used as an example in Section 5.2.1 that states: "For all integers ('a') from zero to one hundred and all arrays ('b') from zero to three length containing integers from negative five to five should follow a particular assertion.".

Table 5.2: First 5 lines of the example scenario's default file

a	b	time	result
0	[]	0.012750999999752821	true
0	[0 0 0]	0.00036799999980384	true
0	[-10 -10 -10]	0.00020899999980324	true
0	[10 10 10]	0.00425099999754321	true
0	[4 2 1]	0.00023100000010384	true

In Listing 5.1 an example of a filter function for the same scenario is presented. The filter function ignores test cases with an 'a' value higher than ten and changes the names of the columns. The first ten lines of the filtered file can be seen in Table 5.3 (p. 34).

```
1 function filter(tc, execT, result) {
2     if (tc.a > 10)
3         return undefined
4     return {column1: tc.a, column2: tc.b, column3: execT, column4: result}
5 }
```

Listing 5.1: Example filter function

Table 5.3: First 5 lines of the example scenario's filtered file

column1	column2	column3	column4
0	[]	0.012750999999752821	true
0	[0 0 0]	0.000367999999980384	true
0	[-10 -10 -10]	0.000208999999980324	true
0	[10 10 10]	0.004250999999754321	true
0	[4 2 1]	0.000231000000010384	true

5.2.5 Input space coverage

When a developer creates a scenario, arbitraries are also created. Because arbitraries are frequently infinite if not limited, the developer usually creates boundaries on which the values generated by the arbitraries should land, which are called input spaces. When a scenario is executed, each arbitrary will generate a number of values equal to the sample size defined by the strategy (FluentCheck's strategy uses a default sample size of one hundred, but this value is configurable), which is many times not enough to cover all of the input space. This functionality sets out to offer the developer a percentile value resembling the amount of input space covered by the arbitraries and the whole scenario. A scenario's input space refers to all the possible combinations of all the values present in each arbitrary's input space.

To enable the display of the input space coverage of each arbitrary and the scenario, the method `withInputSpaceCoverage()` called from the object returned by `fc.statistics()` must be used (or `withAll(csvPath?, csvFilter?)` which encompasses this functionality as well as the output of information when the analyzed property is deemed satisfiable explained in Section 5.2.1, and the test case output functionality explained in Section 5.2.4).

The input space coverage is calculated by taking the number of unique values generated for each arbitrary and for the scenario and dividing them by the size of the associated input space. The scenario's input space size is calculated by multiplying the size of each of the scenario's arbitraries' input space sizes. The calculation of the size of the input space for each default arbitrary is as follows.

Integer

It is calculated by simply calculating the difference between the lower bound and the upper bound declared by the developer plus one.

Real

A real arbitrary's input space is a little more complex because whatever interval the developer sets, there is always an infinite number of real numbers in it. For the input space size of real arbitraries to be calculated, a precision must be specified. The default precision is three,

but it can be changed by the developer using the method `withPrecision(precision)` called from the object returned by `fc.statistics()`. The input space is then calculated by multiplying the difference between the lower bound and the upper bound declared by the developer, multiplying it by ten to the power of the precision plus one. For example, using a precision of three for real values generated from zero to one, the input space would equal one thousand and one hundred. Values that differ on a decimal place over the precision are considered the same for input space size calculation purposes.

Boolean

The input space size is two since the arbitrary can only generate two values.

Array

The simplest way to get the input space size is by calculating the sum of the number of arrays that can be created with each length (from the lower to the upper bound specified by the developer) which equals the size of the input space of the arbitrary used to generate the array values to the power of the length of the array. However, the way it is done is by calculating the geometric series until the maximum length and subtracting the geometric series until the minimum length. A geometric series is the sum of the numbers in a geometric progression, a sequence of non-zero numbers where each term after the first is found by multiplying the previous one by a fixed, non-zero number.

Char

It is calculated by simply calculating the difference between the ASCII [22] code of the start character and the last character declared by the developer plus one.

String

The string arbitrary works as an array arbitrary that uses a character arbitrary to generate its values, so the way to calculate the input space size is the same.

Set

It is calculated by adding all the combinations possible. For example, for a set with four elements, the size of the input space would be calculated like this: $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4}$

Union

It is calculated by adding the input space sizes of all the arbitraries present in the union.

Tuple

It is calculated by multiplying the input space sizes of all the arbitraries present in the tuple.

5.2.6 Graphs

This subsection goes into detail on the implementation of the visual representation of input spaces. Section 5.2.6.1 explains the default graph generation, while Section 5.2.6.2 explains the custom graph creation.

5.2.6.1 Default Graphs

A visual method is usually more intuitive to better understand the distribution of the test cases in the input space. To achieve higher insight, the method `withDefaultGraphs(path?)` called from the object returned by `fc.statistics()` was created and made, so an SVG file named after the arbitrary containing a graph is generated for each specified arbitrary. This method receives as an optional parameter the path in which to create the graphs. For array, set, and string arbitraries (or unions of these types), bar graphs are created representing the number of inputs generated with each length. For every other arbitrary, the graphs present a one dimension indexed visualization of the distribution of the generated values in the input space.

Examples of the two types of default graphs are presented in Figure 5.3 and Figure 5.4. The graphs are relative to the arbitraries of the scenario used as an example in Section 5.2.1 that states: "For all integers (’a’) from zero to one hundred and all arrays (’b’) from zero to three length containing integers from negative five to five should follow a particular assertion.".

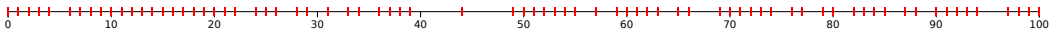


Figure 5.3: Default 1D graph for `fc.integer(0,100)`

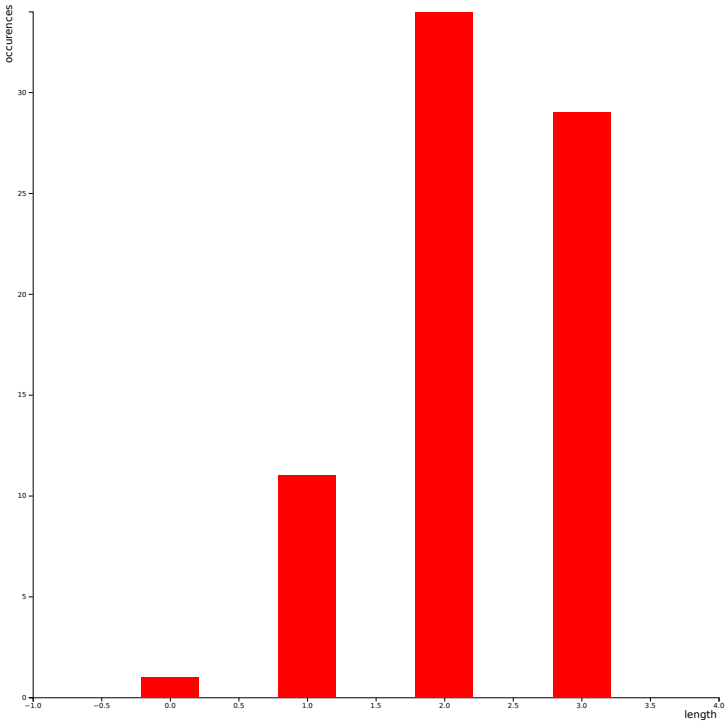


Figure 5.4: Default bar graph for `fc.array(fc.integer(-5,5), 0, 3))`

5.2.6.2 Custom Graphs

To give the developer the ability to configure graphs to their necessity, the `with1DGraph(function, path)` and `with2DGraph(function, path)` methods were created which are called from the object returned by `fc.statistics()` and create custom graphs. The first one generates graphs with a single axis, while the second generates graphs with two axes ('x' and 'y'). As for the parameters, the second one is the graph's path, and the first one is the indexing function which is applied to each generated input and receives as parameters an object containing the values generated for each arbitrary, an object containing the input space size of each arbitrary, the execution time of the run and the run's result. The indexing function's return should be an object whose structure depends on the graph being generated:

1D graphs

value: the place of the entry in the single axis.

color?: the color of the entry in the graph.

2D graphs

valueX: the place of the entry in the 'x' axis.

valueY: the place of the entry in the 'y' axis.

color?: the color of the entry in the graph.

If 'value' in 1D graphs or either of the values in 2D graphs is left undefined (or the return itself is undefined), the entry is ignored and not placed in the graph. The last parameter is the path to which to create the graph (which must contain the file's name in the end without the SVG extension). The graphs also represent the entries in a lighter or darker color depending on the number of repeated values it contains (if all values have the same color strength, they all have the same amount of repeated values). In order to output the graphs in case of a successful run, `withOutputOnSuccess()` must be called (or `withAll(path?, filter?)` because it incorporates the function).

Listing 5.2 contains an example of a custom indexing function to generate a one dimension graph which is represented in Figure 5.5 (p. 38). This custom graph contains the values generated by 'a' but colors the values higher than 50 in blue (the others are red, which is the default color).

```
1 function f1(tc, sizes, time, result) => {
2     return {value: tc.a, color: tc.a > 50 ? 'blue' : undefined}
3 }
```

Listing 5.2: Custom 1D graph indexing function

Listing 5.3 (p. 38) contains an example of a custom indexing function to generate a two-dimension graph which is represented in Figure 5.6 (p. 38). This custom graph contains only entries with an 'a' value higher than 50. It represents the value of 'a' in the 'x' axis and the length

Figure 5.5: Custom 1D graph

of the array generated by 'b' in the 'y' axis. As noticeable in the graph, values generated more times are represented in a darker color. In comparison, ones that were generated fewer times are represented in a lighter color.

```
1 function f1(tc, sizes, time, result) => {
2     return {valueX: tc.a > 50 ? undefined : tc.a, valueY: tc.b.length}
3 }
```

Listing 5.3: Custom 2D graph indexing function

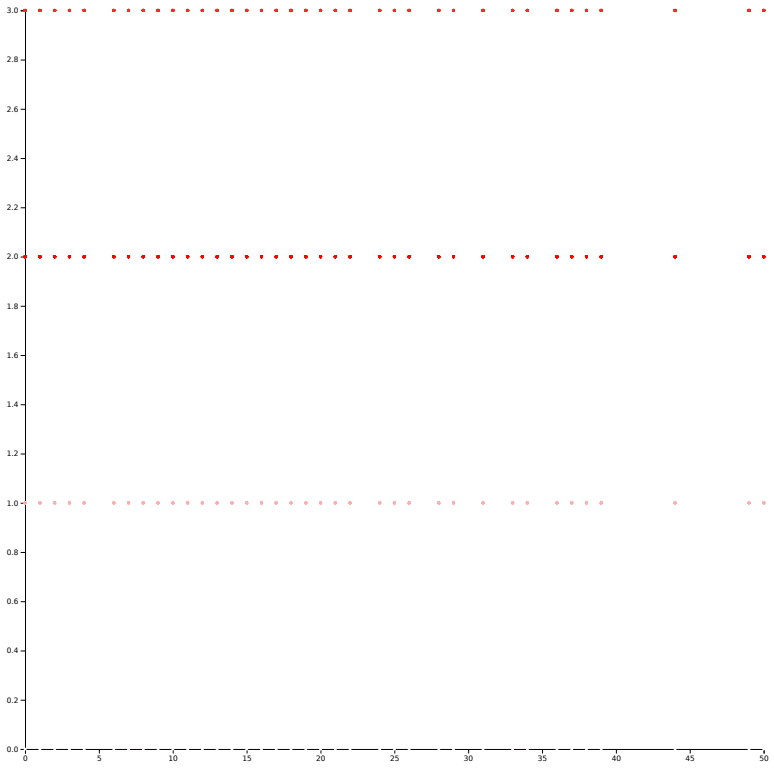


Figure 5.6: Custom 2D graph

5.3 Summary

This chapter described the proposed solution and its implementation in detail. In Section 5.1 a general overview of the solution was given. In Section 5.2, the approach to providing insight on the test case generation process was presented, describing the specific features developed, and the reasoning behind the decisions taken for their development.

The main features developed include:

- Presenting the developer with the ability to specify custom seeded generators and repeating previous executions.
- Providing the developer a configurable file containing all test cases generated, their execution time, and execution result.
- Displaying the input space coverage of not only each specified arbitrary but also the overall scenario.
- Generating graphs representing the test case distribution of each arbitrary and allowing custom user-defined graphs.

Chapter 6

Empirical Evaluation

This chapter presents the empirical evaluation of the developed tool. First, the methodology followed in this study is described in Section 6.1. Then, in Section 6.2, the results, their analysis, and a brief discussion are presented. The threats to validity are enunciated in Section 6.3. Lastly, Section 6.4 summarises the contents of this chapter.

6.1 Methodology

Experimental evaluation plays a vital role in science, enabling the validation of the proposed hypothesis. Due to this information, an empirical study was conducted to validate the hypothesis proposed in Section 4.2. This study evaluated the participants' performance when using the implemented feature and comparing it to the performance when not using it by measuring various metrics explained in Section 6.1.1.

6.1.1 Plan

Empirical evaluation requires the establishment of various criteria as well as planning of the process to be followed. In this section, the plan defined for the experiments is described.

Participants

The tests use Fluent-Check [32], a typescript library, hence software development experience and familiarity with JavaScript/TypeScript were required for all participants. Familiarity with PBT was desired but not required. This study counted on the participation of 16 students and software developers.

Duration

The estimated duration for the test was 30 to 60 minutes. The interval is significant since people with less experience were expected to take more time to solve each task. The participants were also required to read some tutorials beforehand, with each of them having an estimated duration of 15 minutes.

Environment

Due to the ongoing pandemic during the execution of these experiments, they were conducted one on one remotely. Every participant could use their personal computer and preferred operating system as long as it had the latest Node.js [9] version. The used code editor was Visual Studio Code [24], and participants were required to install a character counting extension [10]. Participants were also requested to record the experiment as a redundancy measure to cover any potential mishap.

Procedure

To compare the participants' performance using the developed feature against others not using it, the sixteen participants were split into two groups of eight participants:

Group A

attempted to solve the tasks while using the various components provided by the developed feature.

Group B

attempted to solve the tasks without the components developed by the feature using only the normal Fluent-Check.

Both groups' participants were given a guide that changed slightly depending on their group, which can be found in Appendix A (p. 61). A tutorial about Fluent-Check was provided to both groups and can be found in Appendix B (p. 67). The ones from *group A* were also given a tutorial about the developed feature present in Appendix C (p. 75). The various code files given to the participants are present in Appendix D (p. 85).

Data

An empirical evaluation requires the observation and collection of data for the validation of the proposed hypothesis to be possible. During the experiment, various aspects were recorded for each task besides the final code solution:

- If the participant was able to find the bugs injected in the task's code;
- The non-redundant properties added and if they are correctly coded;
- The time taken to solve the task;
- The number of characters written;
- The number of times the participant tested the task's properties;

- The number of times that the generation parameters used for a property were changed. The changes taken into account include changes to the arbitraries and the sample size.

For tasks performed with access to the developed feature (*group A*), two aspects were also recorded:

- The number of times the participant opened a generated test cases list;
- The number of times the participant opened a generated graph.

Survey

Various surveys were carried out during each experiment with three different parts, each with a different goal:

- a **background** survey which aimed to assess the level of technical knowledge familiarity with the used technologies and some information about the participant;
- **after the tutorial** was carefully read by the participant, a survey was performed to ascertain whether they understood how Fluent-Check and PBT worked.
- a survey **after each task** was performed where participants rated their confidence that all bugs had been found and the presented properties were sufficient. Participants from *group A* were also inquired on which parts of the feature they found useful and why and whether they believed the feature helped them in controlling and understanding the generation process.

6.1.2 Tasks

This empirical study aims not to evaluate if the ability of the participants to find errors improves with the use of the feature but rather to analyze its impact on the participants' testing process and result confidence. The specifics about each task are described in this section.

Task 1

This task aims to ascertain how the participants fare when provided with the system's code and some properties while also serving as an introduction to PBT and the usage of Fluent-Check to its lower difficulty. The task's provided code can be found in Appendix D.1 (p. 85).

It contains a simple function that is supposed to turn lower case alphabetical characters in a string to upper case using ASCII values, requiring these characters' ASCII code to be subtracted thirty-two. Two properties are already provided to the participant: (1) one that checks for the absence of lower case alphabetical characters in the string and (2) another that checks if the resulting string has the same length as the original string.

The provided code, however, contains two errors:

- In the function, every character in the string is subtracted thirty-two, leading to all elements being changed. This error can easily be found by analyzing the function

or defining a property that checks whether the non-upper case characters remain unchanged;

- In the first property that checks for the absence of lower case alphabetical characters in the string, only strings containing lower case characters are generated, resulting in a flawed test since not all possible inputs are being tested. Changing the generation in this property so the string encompasses all characters would not fail due to the first error because there is no character that, when subtracted thirty-two, results in a lower case character.

Task 2

This task aims to ascertain how the participants fare when provided with the system's code and must fully define its properties. The task's provided code can be found in Appendix D.2 (p. 87).

It contains a quicksort implementation which was fetched from the *guru99*[1] website. The participants are told to remind themselves about how quicksort works before starting the experiment.

The provided code contains a single error. The error is present in the quicksort function and consists of subtracting one from the 'right' value in the second nested conditional statement and the second quicksort recursive call. This subtraction results in the right-most element of a right-side partition not being sorted. This error is more likely to happen in large arrays because more partitions of the array are likely to happen but is still common in arrays containing more than two elements.

6.2 Results

This section presents the results of the executed study. In Section 6.2.1, an analysis of the background of the participants is done. In Section 6.2.2, the various metrics gathered from the tasks are analyzed. In Section 6.2.4, the research questions are answered.

For all null hypothesis tests used in this section, the significance level used to compare the p-value was set at 0.05.

6.2.1 Background

The background survey given to every participant was designed to assess the participant's comfort level with the technologies used during the study. In addition, another survey was filled after the participants finished reading the given tutorials and was designed to assess if they understood how Fluent-Check works. Since they contain a set of Likert-type items numbered as follows, this comfort level can be assessed using a Likert-type scale.

B1: I have experience using Visual Studio Code.

[1] https://www.guru99.com/quicksort-in-javascript.html (Retrieved: 21/06/2021)

B2: I have experience using javascript.

B3: I have experience testing software.

B4: I have experience using Mocha.

B5: I have experience using property-based testing.

B6: After reading the tutorials, I believe I understand how property-based testing and FluentCheck work.

Each participant's comfort level was calculated by summing the answers to each item on a scale of -2 (*Strongly Disagree*) to 2 (*Strongly Agree*). Since there are six items in the background survey, the minimum possible value is -12, and the maximum is 12. In order to analyze the comfort levels, the histograms of both groups are present in Figure 6.1.

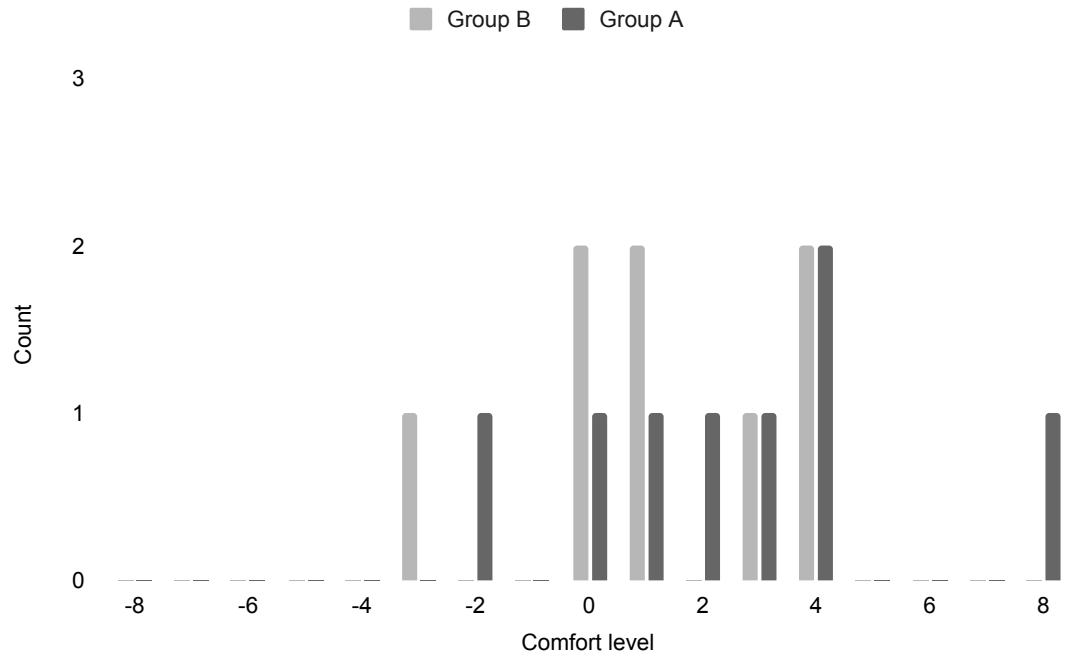


Figure 6.1: Histogram of comfort levels of both groups

By analyzing the histogram, no significant difference can be noticed between both groups besides one outlier on each of them.

Figure 6.2 (p. 46) shows a stacked bar chart with each background survey item's frequency for *group A* is present. In Figure 6.3 (p. 46) the same can be noted, but for *group B*. By looking at the *B4* and *B5* columns, most participants lack experience in using Mocha and PBT. Although the lack of knowledge in Mocha should not be critical for this particular experience, the lack of knowledge in PBT signifies one of the leading causes for the validation threat relative to sample diversity explained in Section 6.3.

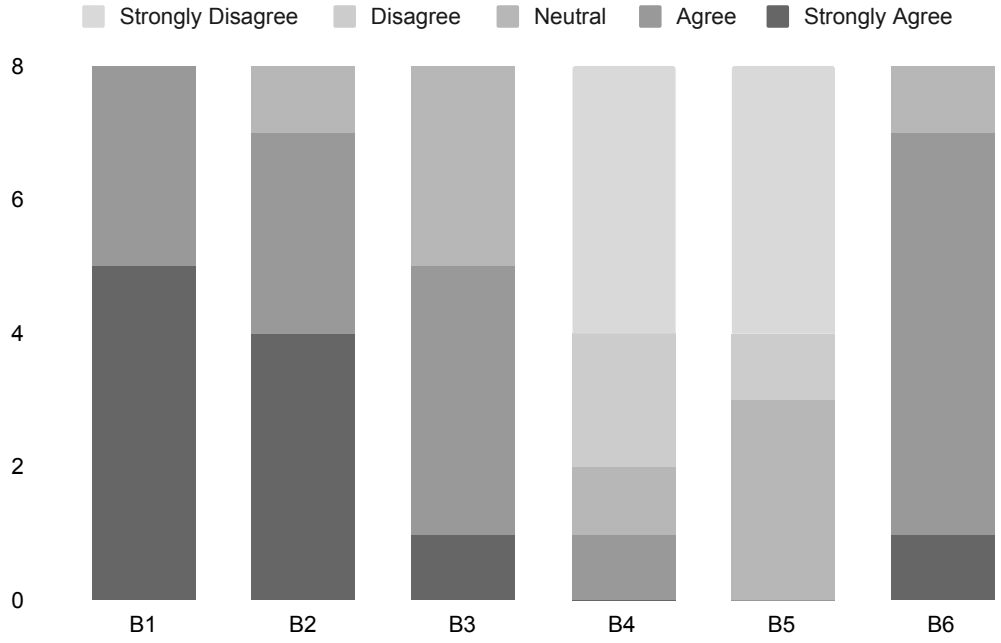


Figure 6.2: Stacked bar chart with frequency of each background survey item - *group A*

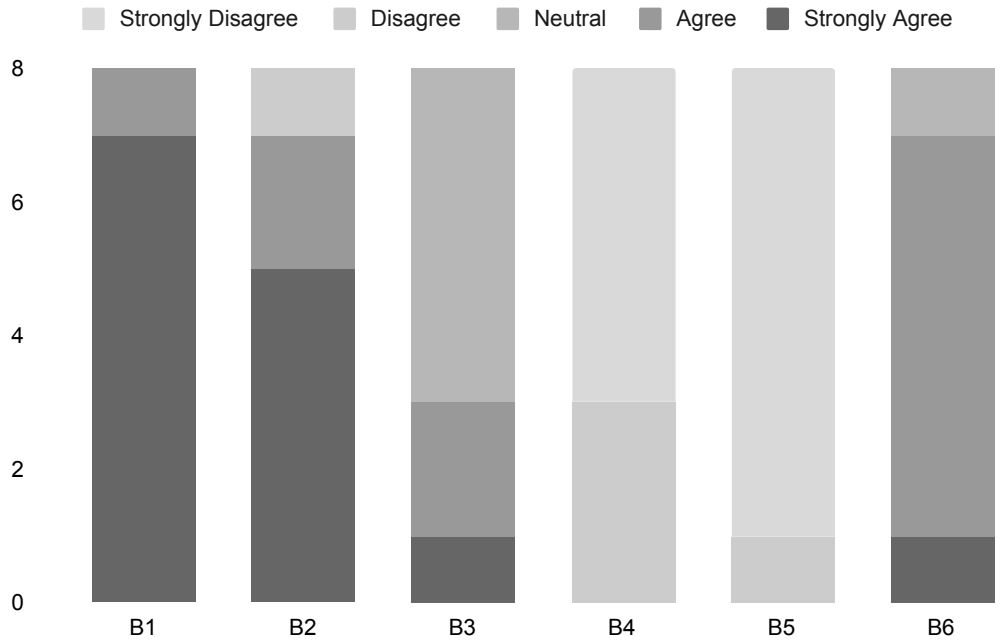


Figure 6.3: Stacked bar chart with frequency of each background survey item - *group B*

Table 6.1: Comfort level statistical measures

Group	Size	Mean	Std. Deviation	Shapiro-Wilk(p)	Levene(p)	t-test(p)
A	8	2.50	3.02	0.98	0.60	0.37
B	8	1.25	2.38	0.46		

Table 6.1 shows some of the statistical measures obtained from the Likert-type scale.

In order to conclude whether or not there are statistical differences between both groups, a hypothesis test is needed. Therefore, the *Student's t-test* was chosen to test the hypothesis that both groups have identical means.

H0: The means of the populations from which each group was sampled are the same.

H1: The means of the populations from which each group was sampled are different.

The common assumptions made when performing a *Student's t-test* include those regarding the measurement scale, random sampling, normality of data distribution, adequacy of sample size, and equality of variance in standard deviation. When looking at the measurement scale, one can conclude that it is correct since Likert-type scales are considered at least ordinal. The random sampling and adequacy of sample size assumptions are covered in Section 6.3 where the threats to validity are addressed. However, it has been shown by various studies that this test can handle small sample sizes[5] and the violation of at least one assumption[15].

In order to test using *Student's t-test*, the assumptions relative to the normality of data distribution and equality of variance in standard deviation must be addressed. This can be done using *Levene's test for equality of variances* and *Shapiro-Wilk test*, which respectively test the null hypothesis that both groups are from populations with equal variances and that each sample comes from a population with a normal distribution. Since the resulting *p-values* present in Table 6.1 are well above the significance level of 0.05 for both tests, the null hypotheses cannot be rejected.

As a consequence of these two tests, it is assumed that both samples come from normally distributed populations, and their variances are equal, allowing the Student's t-test to proceed and test the previously set hypotheses. After calculating the *Student's t-test*, the *p-value* obtained is 0.59, well above the significance level of 0.05, therefore, the Null Hypothesis fails to be rejected. It must then be accepted that there is no statistical difference between the two groups.

6.2.2 Tasks

From the performance of both groups in both tasks performed, various metrics were gathered and described in this section. Section 6.2.2.1 analyzes the number of times the generation of a property was changed to find any statistical difference between both groups.

From the gathered metrics, the first six are quantitative, while the other three are qualitative, which requires both data to be handled differently. Not all data presented here will be analyzed in-depth, but it is displayed for future work purposes.

M1: Time taken to solve the task.

M2: Characters written during the experiment.

M3: Number of times the properties were executed.

M4: Number of times the generation of a property was changed after the first definition.

M5: Times the test case list provided by the feature was opened.

M6: Times the graphs provided by the feature were opened.

M7: The bugs found by the participant.

M8: Valid properties created by the participant.

M9: Correctly coded valid properties.

Table 6.2 displays all of the gathered quantitative data's means and standard deviations. The data is organized by task and group for organization purposes. The metrics are also identified by the identifier previously set for them in this section. Metrics *M5* and *M6* only apply to *group A*.

Table 6.2: Quantitative metrics gathered from each group and task

Metric	**Task 1**				**Task 2**			
	Group A		**Group B**		**Group A**		**Group B**	
	Mean	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**	**Mean**	**Std. Dev.**
M1	26:57	10:26	18:15	04:36	28:43	05:30	26:03	10:00
M2	453.63	253.18	393.38	213.46	548.88	271.66	537.75	223.59
M3	8.00	3.33	2.38	2.07	10.00	5.76	7.75	4.06
M4	2.75	2.66	0.88	0.64	2.88	3.14	1.00	1.31
M5	6.13	3.18			5.63	4.75		
M6	1.75	1.28			1.13	1.64		

Figure 6.4 (p. 49) shows a histogram that analyzes information about metric *M7* and shows how many participants of each group found and fixed each bug implemented in the code.

Table 6.3 (p. 49) shows information about metrics *M8* and *M9*, stating how many participants of each group added each property and how many of those implemented them correctly. The various properties that the participants created are the following:

P1: Non-lower case alphabetical characters remain unaltered.

P2: Resulting array has the same length as the original.

P3: Resulting array has the same elements as the original.

P4: Resulting array is ordered.

P5: Function outputs an array similar to an already implemented sort function.

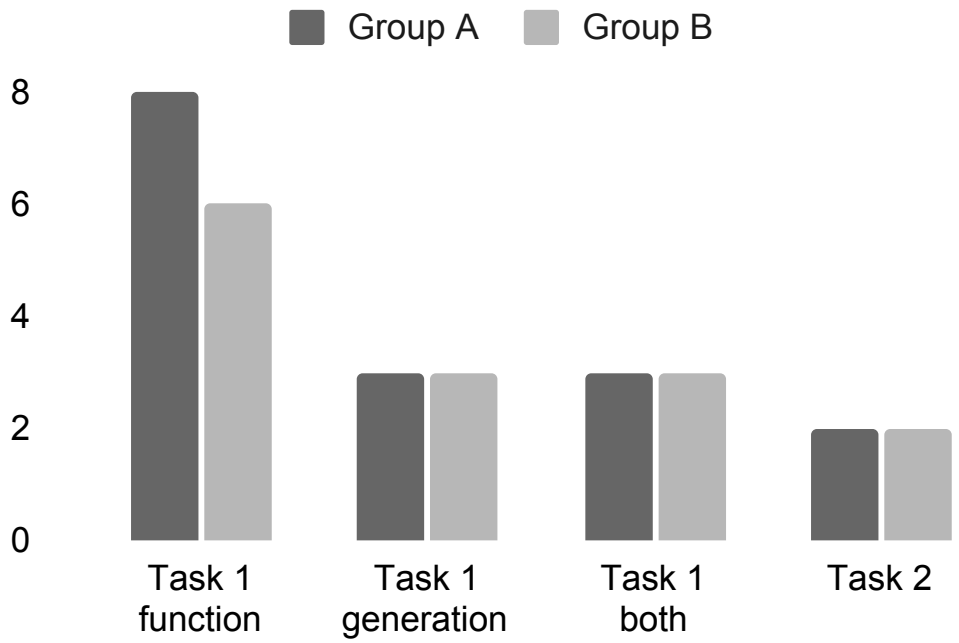


Figure 6.4: Histogram of the number of participants that found each bug by group (*M7*)

Table 6.3: Participants that added each property (*M8*) and correctly coded it (*M9*)

Task	Property	Group A		Group A	
		Added	Correctly coded	Added	Correctly coded
1	P1	4	3	6	4
2	P2	4	4	5	5
	P3	2	2	2	1
	P4	3	3	5	4
	P5	4	2	2	0

6.2.2.1 Generation changes

In Table 6.4, it is possible to compare the mean number of times the generation of a property was changed after the first definition (*M4*) between the group using the proposed feature against that of the other group for each of the tasks.

Table 6.4: Number of times the generation of a property was changed (*M4*)

Task	Group	Mean	Standard Deviation	Mann-Whitney(*p*)
1	A	2.75	2.66	0.14
	B	0.88	0.64	
2	A	2.88	3.14	0.25
	B	1.00	1.31	

In order to understand whether the difference between the two groups is statistically significant, a two-sided *Mann–Whitney U test* was performed to test the following hypotheses:

H0: The population distributions from which each group was sampled are identical.

H1: The population distributions from which each group was sampled are not identical.

Since the resulting *p-values* present in Table 6.4 are above the previously set significance level of 0.05 for both tasks, the null hypotheses cannot be rejected, hence the conclusion that there is not enough evidence to conclude that there are statistical differences between both groups.

6.2.3 After task survey

A survey was given to every participant after each task, and its data is analyzed in this section. It was designed to assess the participant's confidence in the created tests and for elements of *group A* to also provide their opinion about the developed feature. The items are all different in the way the participant expresses them and are described ahead. Section 6.2.3.1 analyzes the participant's confidence in the created tests to find any statistical difference between both groups.

T1: Rate how confident you are that the created properties are sufficient and all bugs have been found.

T2: I believe that the features helped me understand and control the distribution of input values.

T3: Select the features you found useful in order to solve this task.

T4: Explain how the features you found useful helped you in solving the task.

As previously mentioned, *T2*, *T3*, and *T4* only apply to *group A* participants. *T1* is analysed in Section 6.2.3.1 and Figure 6.5 shows a stacked bar graph containing the answers to *T2* which is a Likert-type item. By analyzing the answers, 62.6% of the participants found the proposed feature to help them understand and control the distribution of input values in the task in question against 6.3% who did not find it helpful.

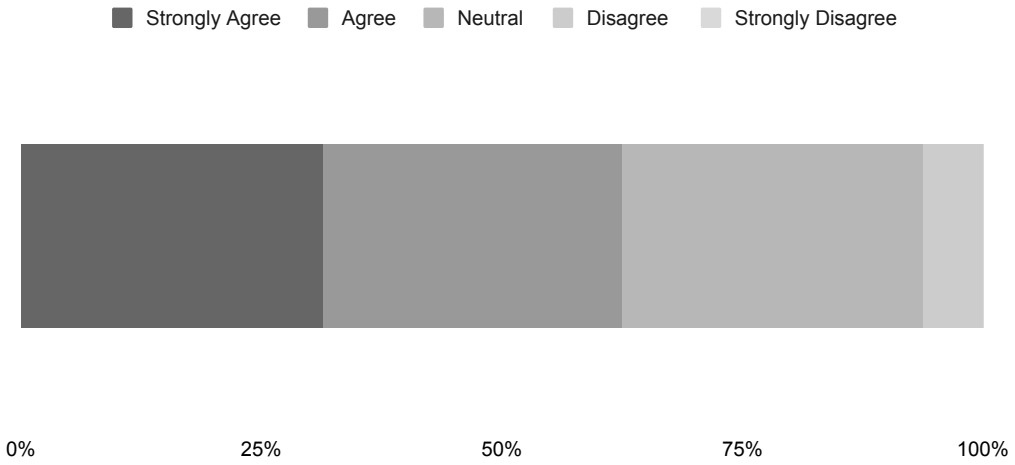


Figure 6.5: Stacked bar graph containing answers to *T2*

Figure 6.6 displays a bar graph where the number of tasks where the participants found each feature helpful is displayed (*T3*). As can be seen, every participant found the test case list to help them in their testing process during every task. However, the same cannot be said for the other features, which show little engagement. This can be due to the lack of experience in PBT that the samples show since they require more knowledge of PBT.

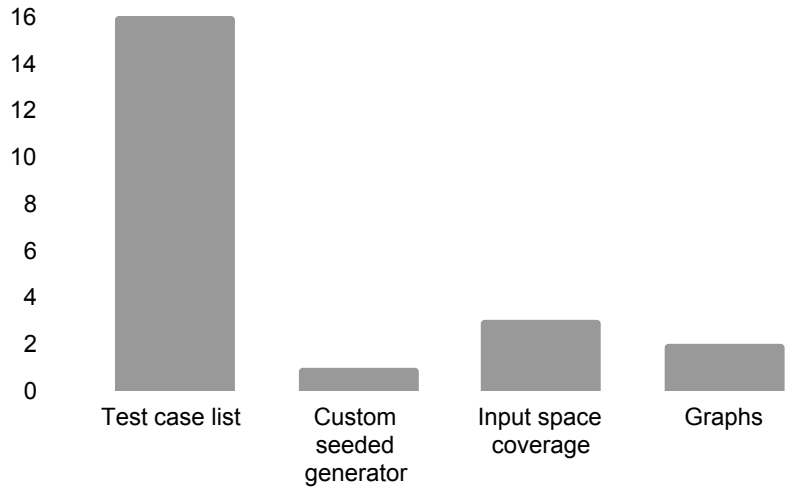


Figure 6.6: Bar graph containing the answers to *T3*

By looking at the answers to *T4*, it can be concluded that some participants found the test case list helpful in order to find patterns between the failing inputs, and others found it helpful in realizing how the generation was occurring. However, it can also be concluded that the participants did not completely understand what the input space coverage value meant and its use. Concerning the other two features, not much was said given the low amount of participants that found it helpful.

6.2.3.1 Confidence in a positive result

In Table 6.5, it is possible to compare the mean participant confidence in the test's positive result between the group using the proposed feature against that of the other group for each of the tasks. The participants of each group that match the requirements for this comparison are the ones that were able to fix the errors in the functions and, in the end, obtained a positive result.

Another comparison could be made between the participants' confidence from each group that accepted a positive result that resulted from insufficient properties that did not detect the errors in the tasks. The sample sizes are, however, tiny, hence no viable conclusion could ever be taken. This situation occurred for one participant of *group B* in *task 1* and one participant of *group A* and two of *group B* in *task 2*.

Table 6.5: Confidence of participants in a positive test result

Task	Group	Sample size	Mean	Std. Deviation	Mann-Whitney(p)
1	A	8	6.88	0.99	0.56
	B	6	5.50	3.39	
2	A	2	9.00	1.41	> 0.05
	B	2	8.00	0.00	

In order to understand whether the difference between the two groups is statistically significant, a two-sided *Mann–Whitney U test* was performed. It was calculated to test the following hypotheses:

H0: The population distributions from which each group was sampled are identical.

H1: The population distributions from which each group was sampled are not identical.

As seen in Table 6.5, the obtained *p-value* is well above the previously set significance level of 0.05, therefore, the Null Hypothesis fails to be rejected. It must then be accepted that there is not enough evidence to conclude that there are statistical differences between both groups.

For *task 2*, since the number of participants that were able to fix the bug correctly is so tiny for both groups, the use of *Mann–Whitney U test* is out of the question since when the total sample size is seven or less, the test will always give a *p-value* greater than 0.05 no matter how much the groups differ [25]. Although there is no minimum sample size for the *Student's t-test* to be valid, it

must be large enough to calculate the test statistic. It is then impossible to conclude whether there is a statistical difference between both groups in the case of *task 2*.

6.2.4 Discussion

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

H1: An informed testing process in PBT leads to higher control over input value distribution and higher confidence in positive run results.

With it, it was expected for the following research questions to be answered:

RQ1: *Does an informed testing process in PBT increase confidence in the positive result of a run?*

The answer to **RQ1** lies in Section 6.2.3.1, where it is concluded that for participants that were able to fix the errors in the functions and, in the end, obtained a passing test, there is **not enough statistical evidence** to conclude that there is an increase in confidence, hence not validating that an informed testing process increases confidence in the positive result of a run. This conclusion relies only on subjective data and does not consider objective metrics because no relation between the gathered metrics and this matter could be found. One can argue that multiple changes in the generation process of a property result from a participant's desire to increase his confidence. However, as concluded in Section 6.2.2.1, no statistical difference between both groups was found.

RQ2: *Does an informed testing process in PBT increase insight and control over the generation of input values?*

In Section 6.2.2.1, it is concluded that there is **not enough statistical evidence** to say that there is an increase in the number of times the generation of a property was changed after the first definition, hence the conclusion that even if an informed testing process increases insight and control over the generation of input values, its effect on the participants' testing process is not statistically sufficient. Section 6.2.3, however, analyzes the participants' answers to the survey given after each task and, one of the items pertains to the participants' opinion on the proposed feature's contribution to their understanding and control of the distribution of input values. A majority of the participants concluded that the proposed feature did help them in this matter (62.6%). However, these results can be skewed due to a desire by the participants to provide a more positive evaluation of the feature.

Overall, the research points towards a lack of evidence that an informed testing process increases the confidence in positive run results and control over input value distribution, thus not validating the hypothesis.

6.3 Threats to validity

Several threats to validity must be considered when conducting an empirical study. In this section, various threats to validity are described grouped by their type.

6.3.1 Construct validity

The first type of validity considered is Construct Validity, which looks at how well the tasks, environment, data collected, and other variables in an empirical study map the constructs or hypothesis they represent [4].

Tasks' validity

The likeliness of the tasks used in the experiment representing faulty programs that developers are likely to find in their daily work is one of the threats to construct validity. Some of the code used was fetched from existing outlets instead of tailored for this specific experience to minimize this problem.

Hypothesis guessing

Some participants might try to deliberately or unknowingly guess the hypothesis and change their process to do better. It is not possible to be sure that a participant is aiming towards a specific result, so the hypothesis and objectives of the study were concealed until the end to minimize the effect of this threat.

Preemptive error discovery

Some participants might find the errors in the tasks before creating any properties that signal their existence. Unfortunately, this possibility cannot be controlled since doing it would imply altering the participant's original process resulting in potentially altered results.

6.3.2 Internal validity

The second type of validity considered is Internal Validity, which concerns the extent to which a valid cause-effect relationship can be noticed between the independent variables and the effects observed in the dependent variables [35].

Paticipants' motivation

Due to the length of the experiments, many of the participants may feel increasingly tired during the process. This may lead to decisions being taken more lightly than they would otherwise. The original plan for the empirical study included a third task where the participants were required to test a function and code it. However, after a training experiment, the participant expressed some tiredness by the end of the second task, hence removing the third one. This aspect that was not included is part of possible future work. Another measure used to minimize the effect of this threat was for the participants to read the required tutorials beforehand, not executing the tasks right after. This allowed the participants not to get tired so quickly during the experiment.

Development environment

Due to the ongoing pandemic during the execution of these experiments, they were conducted one on one remotely. Every participant could use their personal computer and preferred operating system as long as it had the latest Node.js version. This might have led to slight differences in the time required to solving the tasks.

Physical environment

Due to the ongoing pandemic during the execution of these experiments, they were conducted one on one remotely. All the participants were in their own house and were requested to remain in an isolated space while participating in the experiment. There were, however, some interruptions that rarely occurred and that could have had an impact on the result of each task.

6.3.3 External validity

The final type of validity discussed in this section is External Validity, which concerns the degree to which the findings of a study can be generalized to different populations or settings [35].

Sample size

The empirical experiment conducted counted on the participation of 16 students and software developers, which were split into two groups with eight members each. This is a relatively small sample size, which must be considered when interpreting this study's findings.

Sample diversity

When looking at a sample's quality, its composition is of importance. Of the 16 people that participated in this study, five were second-year students, one was third-year, two were fourth-year, six were last-year, and 2 were professionals. However, the findings from a study conducted by Salman *et al*. showed that students perform similarly compared to software engineering professionals when using new technologies during experimentation [31], as is the case here. There were, however, a lack of participants with experience in PBT, and future experiments should attempt to introduce more participants who possess this trait.

6.4 Summary

This chapter presents the empirical evaluation of the developed tool. In Section 6.1 the methodology followed in this study is described.

Then, in Section 6.2, the results of the study are presented, along with a descriptive analysis and an explanation for the methods used, followed by a discussion of these results and an interpretation of the findings.

The threats to validity are enunciated in Section 6.3 along with the steps taken to minimize them.

Chapter 7

Conclusion

This chapter presents a reflection on the work presented in this document. Firstly, in Section 7.1 the conclusions of this work are presented. Then, in Section 7.2, its main contributions are mentioned. Lastly, possible future work regarding a more informed testing process is mentioned in Section 7.3.

7.1 Results

This work started with exploring the state of the art in property-based testing, also known as PBT, and the information provided by the technologies. A feature was proposed to provide a solution to the lack of transparency offered by the present technologies. The formulated hypothesis was as follows:

Hypothesis: An informed testing process in PBT leads to higher control over input value distribution and higher confidence in positive run results.

In order to validate the proposed hypothesis, a feature for the PBT library Fluent-Check [32] written in Typescript was developed where various metrics about the generation process were offered to the developer. Then, an empirical study was performed in which the confidence perceived by the participants and their control of the generation process was evaluated.

This experiment was used to validate the hypothesis mentioned above and answer the following research questions:

RQ1: *Does an informed testing process in PBT increase confidence in the positive result of a run?*

It was concluded that there is **not enough statistical evidence** to conclude that there is an increase in confidence in participants presented to the feature. This conclusion relies only on subjective data and does not consider objective metrics because no relation between the gathered metrics and this matter could be found.

RQ2: *Does an informed testing process in PBT increase insight and control over the generation of input values?*

It was concluded that there is **not enough statistical evidence** to say that an informed testing process increases insight and control over the generation of input values. However, analysis to the survey provided after each task where participants expressed the perceived usefulness of the feature for their control and understanding of the generation process revealed that a **majority of participants found it helpful** (62.6%). However, these results can be skewed due to a desire by the participants to provide a more positive evaluation of the feature.

Ultimately, the hypothesis could not be validated. One of the main possible reasons for that is the low sample size, resulting in some statistical uncertainty, especially when responding to *RQ1*.

7.2 Main contributions

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

- A **feature** for the PBT library Fluent-Check written in Typescript, where various metrics about the generation process are offered to the developer to aid decisions relative to the acceptance of positive run results and improve the understanding and control of the input generation mechanism;
- An **empirical study** with 16 participants where half of them used the developed feature, and the others did not. The results from this study are not statistically relevant to conclude an acceptance of the hypothesis, but they do show that a larger sample size might result in a more favorable result.

7.3 Future work

Multiple improvements to the developed feature can be implemented, and different experiments can, and should, be conducted in order to validate it further. Here some possible future work developed in this area is presented:

- Providing a confidence level relative to the likelihood of the generated values representing the entire input space. This was the original objective of this work, but a different approach ended up taking over.
- Conducting experiments with a larger sample and the following characteristics:

- Containing a task where the participants are required to test a function and code it. This task was previously planned to be conducted, but after some training experiments, it was concluded that its inclusion would result in too much tiredness to the participants, compromising the study's validity.
- With a sample more experienced in PBT. The one used was mainly comprised of participants new to the testing process and, although a tutorial was given to them, the expected results are not the same.

Appendix A

Guides

A.1 Group A

Guide

This whole study should take you around one hour.

Before starting

Before starting the study you should do the following in order:

1. answer a simple background survey
2. read the following tutorials which should take you around half an hour:
    - fluent-check - explains how Property-based testing (PBT) and fluent-check work. Fluent-check is the PBT library used in this study.
    - fluent-check statistics feature - explains how to use the statistics feature which is the subject of this study.
3. answer a survey about your knowledge of PBT after reading the tutorial.

It is recommended that you also remind yourself on how quicksort works before the study starts by watching a video (please don't look at any code, simply remind yourself of how it works).

Setup

What you must have installed:

- Visual Studio Code - you are expected to use this code editor.
- Visual Studio Code extension - this extension counts the characters inserted and deleted.
- Node.js - used JavaScript runtime.
- any screen recorder - you should record everything you do since the start of the study and then send it (you must record the whole screen, not only the code editor). You can also record each task individually.

To get access to fluent-check and the tasks (which are explained in the next section), you must extract the contents of the zip file containing them and run **'npm install'**. You should only look at the tasks when you start solving them.

To compile the project you should run: **'npm run prepare'** inside the fluent-check directory.

The tasks' files are in the 'test' folder and to run them you should use **'npm run test'** also in the fluent-check directory.

What to send after the study

After the study you should provide:

- Video of the experience
- Final code of the tasks

Study

The tasks should take you around half an hour to solve.

During this study you will be presented with two tasks:

1. you are expected to debug the provided function and tests along with creating any properties you believe will improve the test;
2. you are required to test the system provided using fluent-check;

The provided code might or might not have errors, it's up to you to decide that!

During the study you have full access to the internet to search for concepts and documentation, please do not compare implementations of any of the task functions to find any possible errors.

You are expected to use the developed feature to aid you in your coding and debuging process! When analyzing a property's statistics, you should comment the configStatistics call of other properties to avoid cluttering the console and the destination folder with csv and svg files (all properties have this option commented by default).

During the task, you should record the whole screen (not only Visual Studio Code) using any software of your choice showing your process. You are expected to frequently check the various outputs of the feature.

After every task you must answer an after task survey.

You will be presented with two tasks:

- **Task 1 - Upper Case** - You have to debug a function along with its properties. Errors can be anywhere, not only in the code, but also in the given properties. You should also create any properties you believe will improve the test. The function takes a string and converts all of its alphabetical characters to upper case. In this task you are expected to work only with ascii values and not use the toUpperCase function. Example: upperCase('*aBc .eL'*) => 'ABC .EL'
- **Task 2 - Quicksort** - You have to test a quicksort algorithm using PBT. Example quickSort([5,1,3,4,7,4]) => [1,3,4,4,5,7]

A.2 Group B

Guide

This whole study should take you around one hour.

Before starting

Before starting the study you should do the following in order:

1. answer a simple background survey
2. read the tutorial about fluent-check which explains Property-based testing (PBT) and how fluent-check work and should take you around fifteen minutes. Fluent-check is the PBT library used in this study.
3. answer a survey about your knowledge of PBT after reading the tutorial.

It is recommended that you also remind yourself on how quicksort works before the study starts by watching a video (please don't look at any code, simply remind yourself of how it works).

Setup

What you must have installed:

- Visual Studio Code - you are expected to use this code editor.
- Visual Studio Code extension - this extension counts the characters inserted and deleted.
- Node.js - used JavaScript runtime.
- any screen recorder - you should record everything you do since the start of the study and then send it (you must record the whole screen, not only the code editor). You can also record each task individually.

To get access to fluent-check and the tasks (which are explained in the next section), you must extract the contents of the zip file containing them and run **'npm install'**. You should only look at the tasks when you start solving them.

To compile the project you should run: **'npm run prepare'** inside the fluent-check directory.

The tasks' files are in the 'test' folder and to run them you should use **'npm run test'** also in the fluent-check directory.

What to send after the study

After the study you should provide:

- Video of the experience
- Final code of the tasks

Study

The tasks should take you around half an hour to solve.

During this study you will be presented with two tasks:

1. you are expected to debug the provided function and tests along with creating any properties you believe will improve the test;
2. you are required to test the system provided using fluent-check;

The provided code might or might not have errors, it's up to you to decide that!

During the study you have full access to the internet to search for concepts and documentation, please do not compare implementations of any of the task functions to find any possible errors.

During the task, you should record the whole screen (not only Visual Studio Code) using any software of your choice showing your process.

After every task you must answer an after task survey.

You will be presented with two tasks:

- **Task 1 - Upper Case** - You have to debug a function along with its properties. Errors can be anywhere, not only in the code, but also in the given properties. You should also create any properties you believe will improve the test. The function takes a string and converts all of its alphabetical characters to upper case. In this task you are expected to work only with ascii values and not use the toUpperCase function. Example: upperCase('*aBc .eL'*) => 'ABC .EL'
- **Task 2 - Quicksort** - You have to test a quicksort algorithm using PBT. Example quickSort([5,1,3,4,7,4]) => [1,3,4,4,5,7]

Appendix B

Fluent-check tutorial

Fluent-check tutorial

Property-based testing

Property-based testing, commonly referred to as PBT, combines the specification of properties with random input generation (fuzzing) to assure they are true. Inputs are generated in a defined way and tested following each property's code.

A property is basically an aspect of a system that, together with other properties, define its mechanics. Finding which properties to use can often be a challenge. A common mistake that developers make when first engaging PBT is defining redundant properties which is common since it is hard to know when the properties specified are enough to define the system in question.

Fluent-check's fuzzing consists of randomly generating unique values and testing the defined properties using these values.

To show the importance of property selection, let's take addition as an example. Three of its properties are the commutative property (add(a,b) == add(b,a)), the associative property (add(add(a,b),c) == add(a,add(b,c))), and the closure property (in 'add(a,b) = c' if type of 'a' equals type of 'b' then type of 'c' must be the same) but these alone are not enough to represent addition. Other operations have these properties such as multiplication. If the additive identity property (add(a,0) = a) is added, the properties are no longer properties of multiplication. They are also, however, properties of XOR. To differentiate addition from XOR, one must add another property, an extra property stating that: sum(a,b) == sum(a-1,b+1). So, as shown, it is complicated to define something so correctly that its definition does not fit that of anything else.

Example

In example.test.ts present in the test folder of fluent-check, a sum function and its properties are specified following the explanation given in the PBT section of the guide. Use this example to understand the following sections.

Comutative property

In this property, 2 integers from -100 to 100 are generated and tested using the commutative property assertion. 10000 input combinations will be provided (100 ^ 2) (explained in scenario creation section).

Associative property

Since 3 different integers are needed to test the associative property, in order to generate around 10000 input combinations like in the other properties instead of 100 ^ 3, the sample size is changed to 21 which results in 21 ^ 3 values being generated which results in around 9261 inputs.

Identity property

```
it('Comutative property of addition', () => {
  fc.expect(fc.scenario()
    .forall('a', fc.integer(-100, 100))
    .forall('b', fc.integer(-100, 100))
    .then(({a, b}) => sum(a,b) === sum(b,a))
    .check()
  )
})
```

```
it('Associative property of addition', () => {
  fc.expect(fc.scenario()
    .config(fc.strategy().defaultStrategy().withSampleSize(21))
    .forall('a', fc.integer(-100, 100))
    .forall('b', fc.integer(-100, 100))
    .forall('c', fc.integer(-100, 100))
    .then(({a, b, c}) => sum(sum(a,b), c) === sum(a, sum(b,c)))
    .check()
  )
})
```

```
it('Identity property of addition', () => {
  fc.expect(fc.scenario()
    .config(fc.strategy().defaultStrategy().withSampleSize(10000))
    .forall('a', fc.integer(-5000, 5000))
    .then(({a}) => sum(a,0) === a)
    .check()
  )
})
```

Since only 1 integer is needed to test the identity property, in order to generate 10000 inputs, the sample size is changed using the config method. The interval of generation for the 'a' variable's arbitrary was changed so that amount of values can be generated (if it was [-100, 100] then only 201 values could be generated).

Closure property

```
it('Closure property of addition', () => {
  fc.expect(fc.scenario()
    .forall('a', fc.integer(-100, 100))
    .forall('b', fc.integer(-100, 100))
    .then(({a, b}) => Number.isInteger(sum(a,b)))
    .check()
  )
})
```

Extra property

```
it('Extra property of addition', () => {
  fc.expect(fc.scenario()
    .forall('a', fc.integer(-100, 100))
    .forall('b', fc.integer(-100, 100))
    .then(({a, b}) => sum(a,b) === sum(a-1,b+1))
    .check()
  )
})
```

Scenario creation

A scenario is used to configure a property's assertion and input generation procedure. To create a new scenario, **fc.scenario()** is used. From its return, you specify the arbitraries used for input generation. Fluent-check works using first-order logic and the arbitraries are specified using the following methods which are called from the scenario:

- config(strategy) - can be used to configure multiple aspects of the scenario. The only expected use by you of this feature is by using this to change the sample size of each arbitrary. To do this simply use the following line and change the number to whatever sample size you want.

  - .config(fc.strategy().defaultStrategy().withSampleSize(50));

- forall(name, arbitrary) - states that for all values generated by this arbitrary, the property must hold (universal quantification);
- exists(name, arbitrary) - states that for at least one value generated by this arbitrary, the property must hold (existential quantification);
- given(name, function) - creates a variable which is defined by the function's return. The function can receive the run value associated to whichever variable was defined previously using "forall", "exists" or "given".
- when(function) - used to make changes on the generated values via a function passed as parameter.
- then(assertion_function) - receives a function that receives as parameters the values generated by the previously set arbitraries and returns a boolean;
- and(function) - can only be used after "given", "when" and "assert" and works the same as the preceding call.
- check() - is used to run the scenario.

For **each arbitrary** associated with the scenario, it will generate **100 values** (can be changed using the "config" method). Since every combination of the generated inputs is provided to the assertion, it will be tested 100 ^ (number of arbitraries) times.

To test the scenario simply use fc.expect(scenario) which prints at the end of a failed run a counterexample. A **counterexample** is an input that causes the system to fail.

Arbitraries

An arbitrary is an abstract class used to wrap information concerning a specific type, whether user-defined or built in the PBT technology. They are used to configure the inputs generated for each property.

Fluent-check contains the following default arbitraries (parameters containing a question mark are optional):

- **integer** - fc.integer(min?, max?) - generates random integers from min to max. If min or max aren't specified, the minimum or maximum safe integers are used.
  - Ex: fc.integer(-500, 500) - generates integers in [-500,500].

- **real** - fc.real(min?, max?) - generates random real values from min to max. If min or max aren't specified, the minimum or maximum safe integers are used.
  - Ex: fc.real(0, 1) - generates real values in [0,1].
- **boolean** - fc.boolean() - generates a random boolean.
- **array** - fc.array(arbitrary, min?, max?) - generates random arrays with elements generated by a specified arbitrary. Min and max can be used to specify the minimum and max length of the arrays. If left unspecified, zero is used as the minimum length and ten as the maximum. Arrays of different lengths all have the same chance of being generated despite the number of inputs that can be generated with that length.
  - Ex: fc.array(fc.integer(-5,5), 0, 3) - generates arrays from length zero to three containing integers from minus five to five.
- **char** - fc.char(start?, end?) - generates a random character from start to end using ascii values. Start and end signify a range of values from the ascii table which can be generated (only generates valid characters). If only start is specified, only that character is generated. If none are specified, every usable character can be generated.
  - Ex: fc.char('a', 'z') - generates lower case alphabetical characters.
- **string** - fc.string(min?, max?, char?) - generates random strings with length from min to max. If left unspecified, min is assigned a value of two, and max a value of 10. To specify the type of characters the string contains, a char arbitrary can be specified (if not specified, it generates all types of characters). Strings of different lengths all have the same chance of being generated despite the number of inputs that can be generated with that length.
  - Ex: fc.string(0, 8, fc.char('a', 'Z')) - generates strings from length zero to eight containing alphabetical characters.
- **set** - fc.set(elements, min?, max?) - generates all combinations of the given elements with lengths from min to max. If min or max are not specified then min is assigned the value of zero and max becomes the number of elements (if the number of elements is higher than ten, max is assigned the value of ten).
  - Ex: fc.set(['a', 'b', 'c', 'd']) - generates all combinations of the characters "a", "b", "c" and "d" from length zero to four.

- **union** - fc.union(...arbitraries) - receives various arbitraries as parameter and randomly chooses one of them to create the input, making arbitraries with a higher number of possible inputs have a higher chance of being generated.
  - Ex: union(integer(0,10), integer(70,100)) - generates integers from zero to ten or integers from seventy to one hundred.
- **tuple** - fc.tuple(...arbitraries) - receives various arbitraries as parameter and generates an array containing a random value of each arbitrary, creating various inputs in a single one.
  - Ex: fc.tuple(fc.integer(-10,10), fc.boolean()) - generates an array containing a random integer from negative ten to ten and a boolean (ex: [-5, false]).

Appendix C

Feature tutorial

Statistics feature tutorial

The objective of this feature is to provide various statistics about the input generation process to aid the developer in better understanding it.

The function fc.expect(scenario) is in charge of receiving the output of an executed scenario and presenting the output to the user. In order to configure the information presented, the user must configure the scenario using the configStatistics(statistics) method.

fc.scenario().configStatistics(fc.statistics())

Fc.statistics() contains the basic configuration which will only present the following information in case of a failed execution: the execution time, seed used in the run (the default generator is not seeded so one muse be specified) and the counterexample.

The counter example is the smallest input that caused the property in question to fail.

To present more information, the following methods can be used:

- **withOutputOnSuccess()** - makes it so information is also presented in the end of a successfull run;
  - Example: **fc.scenario().configStatistics(fc.statistics().withOutputOnSuccess())**
- **withTestCaseOutput(path?, filter?)** - makes it so the generated test cases are presented in the end of the run to a csv file (explained better further ahead in the test case output section);
  - Example: **fc.scenario().configStatistics(fc.statistics().withTestCaseOutput())**
- **withInputSpaceCoverage()** - makes it so the percentage of the possible input space that was generated is presented in the end of the run. The input space refers to all the possible values that can be generated. The input space of integer(0,99) is all the values from zero to one hundred (size = 100). If 50 values are generated then the coverage is 50%;
  - Example: **fc.scenario().configStatistics(fc.statistics().withInputSpaceCoverage())**
- **withAll(path?, filter?)** - makes it so all the 3 previous features are used. Receives as parameters the same as in withTestCaseOutput since it also incorporates its functionality (more about these parameters further ahead);
  - Example: **fc.scenario().configStatistics(fc.statistics().withAll())**
- **withPrecision(precision)** - because real values have an infinite input space, in order to calculate the input space coverage (and to generate the default graphs), a precision must be specified (default is 3 if it is not changed by use of this method);

  - Example: **fc.scenario().configStatistics(fc.statistics().withPrecision(2))**

- **withDefaultGraphs(path?)** - makes it so a graph is generated for each specified arbitrary. For array, set and string arbitraries (or unions of these types) the graphs represent the amount of inputs generated with each length. For every other arbitrary the graphs present an indexed visualization of the distribution of the generated values in the input space. This method receives as an optional parameter the path in which to create the graphs. Examples of default graphs are present in the default graphs section further ahead;

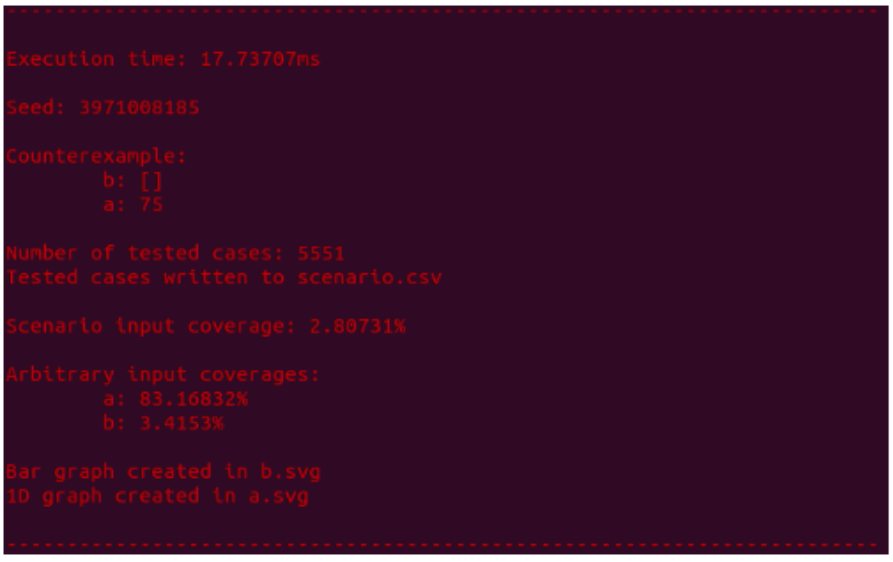
  - Example: **fc.scenario().configStatistics(fc.statistics().withDefaultGraphs())**

- **with1DGraph(function, path)** and **with2DGraph(function, path)** - allows the creation of optional custom graphs (explained better further ahead in the custom graph section).

Information presented in the console

The scenario used for the following examples states: "For all integers ('a') from zero to one hundred and all arrays ('b') from zero to three length containing integers from negative five to five should follow a particular assertion.". The scenario enables all of the features.

Example of information presented in the console in the end of a failed run:



Example of information presented in the console in the end of a successfull run:

```
Execution time: 10.67854ms

Seed: 1234
Number of tested cases: 2500
Tested cases written to scenario.csv

Scenario input coverage: 1.69074%

Arbitrary input coverages:
        a: 49.50495%
        b: 3.4153%

Bar graph created in b.svg
1D graph created in a.svg
```

Custom seeded generator

In order to be able to repeat previous executions, the scenario must use a seeded generator (the default isn't seeded). By using the scenario method withGenerator(generator, seed?) and specifying the desired seed, this is possible. A seeded generator allows the developer to reproduce previous executions, generating the exact same cases. The seed used is presented in the end of the run if the withOutputOnSuccess flag is used.

The generator function must have the following signature: (seed: number) => () => number. Example:

```
let seededGen: (seed: number) => () => number =
  (seed: number) => () => (seed = seed * 16807 % 2147483647) / 2147483647
```

Test case output

The test cases generated are written to a csv file with one column for each arbitrary, one for the execution time, and one for the result of the test case's execution.

As for the available parameters:

1. the path where to write the CSV, which must contain the file's name in the end without the extension and, if left unspecified, writes it to the current directory addidng a new one for each run.
2. an optional filter function which is applied to each generated input. It receives as parameters an object containing the values generated for each arbitrary, the run's execution time, and the result of the run. The filter

should return an object containing keys referring to the CSV column's names and their assigned values for each row. In order to ignore specific executions, an undefined value can be returned, and that row will not be written to the file.

You are not required to use a filter. If you do not use a filter this is an example of a resulting csv:

a	b	time	result
0	[]	0.012750999999752821	true
0	[0 0 0]	0.000367999999980384	true
0	[-10 -10 -10]	0.00020899999981338624	true
0	[10 10 10]	0.00013899999976274557	true
0	[4 2 1]	0.00012299999998031417	true
0	[9]	0.00014300000002073939	true
0	[-6]	0.00013600000011138036	true
0	[2 -5]	0.00013200000001214794	true
0	[-6 -2]	0.00011799999992945231	true

The following example exemplifies the use of a filter which is optional and will create a csv containing only runs that have an 'a' value in [0,10] and the columns: valor_a (containing value of a), valor_b (containing value of b), tempo (containing execution time of the run) and resultado (containing the result of the run).

```
const f = (tc, time, result) => {
  if (tc.a > 10)
    return undefined
  return {valor_a: tc.a, valor_b: tc.b, tempo: time, resultado: result}
}

fc.expect(fc.scenario()
  .configStatistics(fc.statistics().withAll('/home/user/Desktop/filtered_testCases', f))
  .withGenerator(prng, 1234)
  .forall('a', fc.integer(0,100))
  .forall('b', fc.array(fc.integer(-10,10), 0, 3))
  .then(({a}) => a === a)
  .check()
)
```

First 5 lines of the csv created from the example:

1	valor_a	valor_b	tempo	resultado
2	0	[]	0.019096999999419495	true
3	0	[0 0 0]	0.00040499999977328116	true
4	0	[-10 -10 -10]	0.0001449999999749707	true
5	0	[10 10 10]	0.00013800000033370452	true

Default graphs

In this section, the default graphs for the following example property are presented.

```
fc.expect(fc.scenario()
  .config(fc.strategy().defaultStrategy().withSampleSize(75))
  .configStatistics(fc.statistics().withOutputOnSuccess().withDefaultGraphs('/home/user/Desktop'))
  .forall('a', fc.integer(10, 100))
  .forall('b', fc.array(fc.integer(-10, 10), 2, 3))
  .then(({a, b}) => a + b[0] === a + b[0])
  .check()
)
```

Default Graph for ‘a’ (‘a.svg’):

Default Graph for ‘b’ (‘b.svg’):

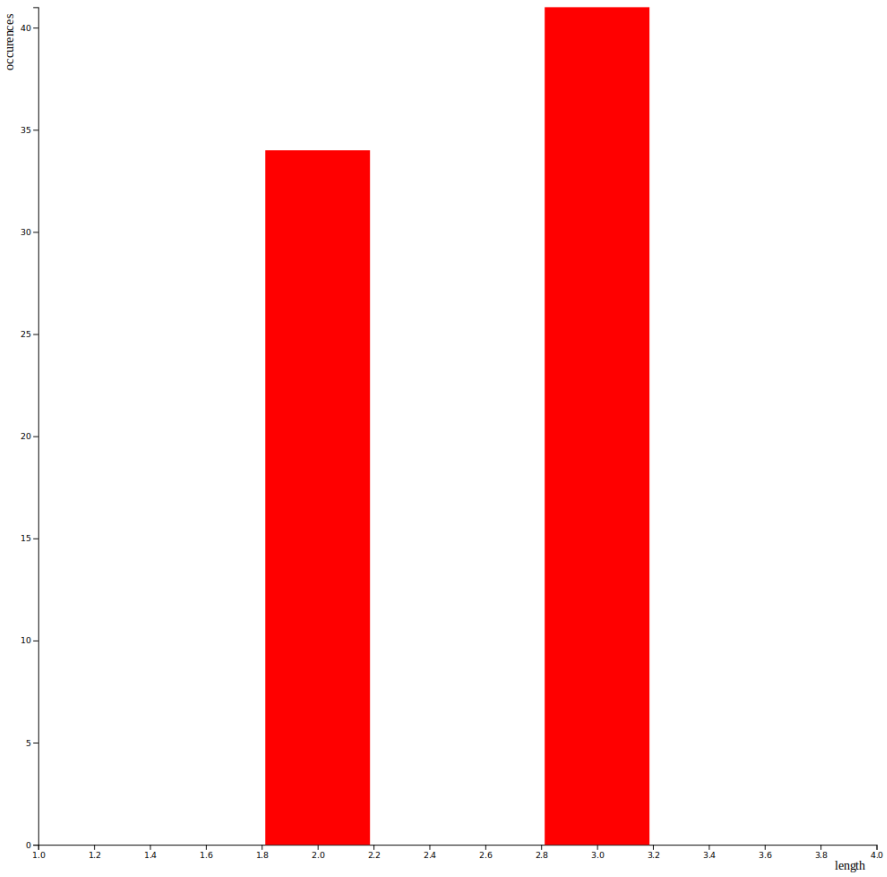
Custom Graphs

Custom graphs are optional and this section explains how to use them.

Using the with1DGraph(function, path) and with2DGraph(function, path) methods you can specify custom graphs. The first one generates graphs with a single axis while the second one generates graphs with two axis (x and y).

As for the parameters, the first one is the indexing function which is applied to each generated input and receives as parameters an object containing the values generated for each arbitrary, an object containing the input space size of each arbitrary, the execution time of the run and the result of the run. The indexing function should return an object containing:

- **1D graphs:**
  - value: place of the entry in the single axis
  - color?: color of the entry in the graph
- **2D graphs:**


occurences
0
5
10
15
20
25
30
35
40
1.0
1.2
1.4
1.6
1.8
2.0
2.2
2.4
2.6
2.8
3.0
3.2
3.4
3.6
3.8
4.0
length

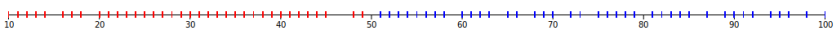
- valueX: place of the entry in the ‘x’ axis
- valueY: place of the entry in the ‘y’ axis
- color?: color of the entry in the graph

If value in 1D graphs or either of the values in 2D graphs is left undefined (or the return itself is undefined) the entry is ignored and not placed in the graph. The last parameter is the path to which to create the graph (which must contain the name of the file in the end without the .svg extension). The graphs also represent the entries in a lighter or darker color depending on the number of repeated values it contains (if all values have the same color strength then they all have the same amount of repeated values). In order to output the graphs in case of a successfull run, withOutputOnSuccess() must be called (or withAll(path?, filter?) because it incorporates the function).

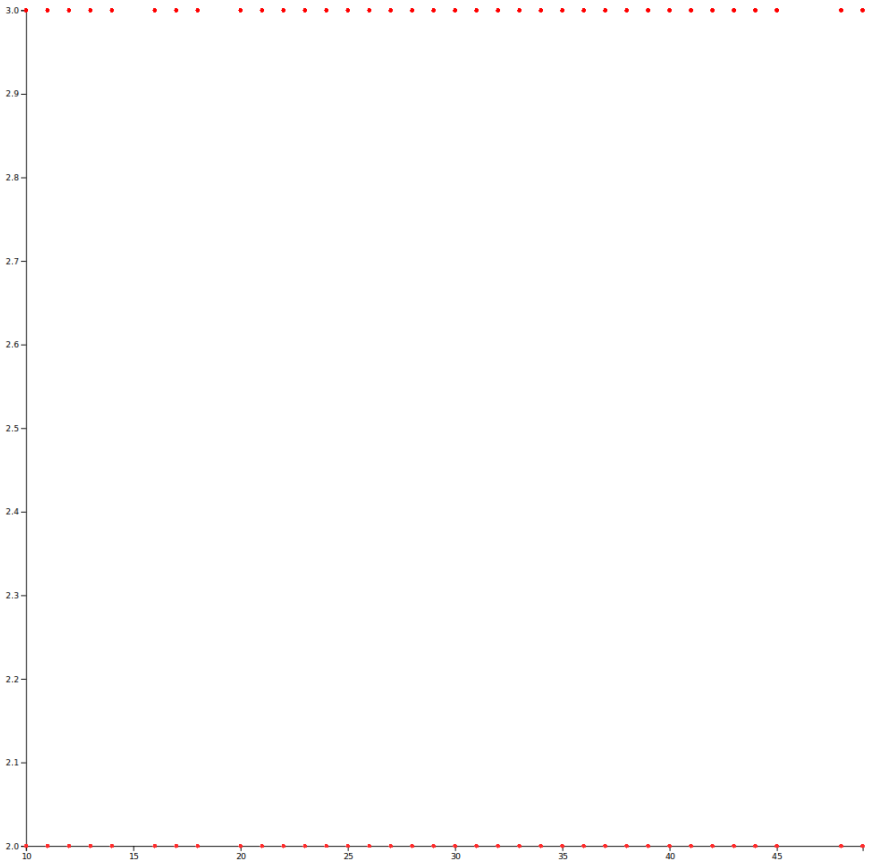
The following example will create a 1D graph which contains the values generated by ‘a’ but colors the values higher than 50 in the color blue (the others are red which is the default color). It also creates a 2D graph that only contains entries with an ‘a’ value higher than 50. It represents the value of ‘a’ in the ‘x’ axis and the length of the array generated by ‘b’ in the ‘y’ axis.

```
const f1 = (tc, sizes, time, result) => {
  return {value: tc.a, color: tc.a > 50 ? 'blue' : undefined}
}

const f2 = (tc, sizes, time, result) => {
  return {valueX: tc.a > 50 ? undefined : tc.a, valueY: tc.b.length}
}

fc.expect(fc.scenario()
  .configStatistics(fc.statistics().withOutputOnSuccess().withDefaultGraphs('/home/user/Desktop')
    .with1DGraph(f1, '/home/user/Desktop/f1').with2DGraph(f2, '/home/user/Desktop/f2'))
  .forall('a', fc.integer(10, 100))
  .forall('b', fc.array(fc.integer(-10, 10), 2, 3))
  .then(({a, b}) => a + b[0] === a + b[0])
  .check()
)
```

1D graph created by the example (f1.svg):



2D graph created by the example (f2.svg):


3.0
2.9
2.8
2.7
2.6
2.5
2.4
2.3
2.2
2.1
2.0
10
15
20
25
30
35
40
45

Appendix D

Provided code

D.1 Task 1

D.1.1 Group A

```
1  import * as fc from '../../src/index'
2  import {it} from 'mocha'
3
4  function upperCase(text: string) {
5    let result = ''
6
7    for (let i = 0; i < text.length; i++)
8      result += String.fromCharCode(text.charCodeAt(i) - 32)
9
10   return result
11 }
12
13 describe('Upper case properties', () => {
14
15   let seededGen: (seed: number) => () => number
16
17   beforeEach(() =>
18     seededGen = (seed: number) => () => (seed = seed * 16807 % 2147483647) /
           2147483647
19   )
20
21   it('Resulting string doesn\'t contain lower case letters', () => {
22     fc.expect(fc.scenario()
23       //.configStatistics(fc.statistics().withAll('uppercase_p1').withDefaultGraphs
             ())    //displays information after a run and also generates csv and
           graphs
24       .withGenerator(seededGen)
25       .forall('text', fc.string(0, 10, fc.char('a', 'z')))
26       .then(({text}) => !/[a-z]/.test(upperCase(text)))     //regex
```

```
      .check()
    )
  })

  it('Resulting string has the same length as original', () => {
    fc.expect(fc.scenario()
      //.configStatistics(fc.statistics().withAll('uppercase_p2').withDefaultGraphs
          ())    //displays information after a run and also generates csv and
          graphs
      .withGenerator(seededGen)
      .forall('text', fc.string(0, 10))
      .then(({text}) => text.length === upperCase(text).length)
      .check()
    )
  })

})
```

D.1.2 Group B

```
import * as fc from '../../src/index'
import {it} from 'mocha'

function upperCase(text: string) {
  let result = ''

  for (let i = 0; i < text.length; i++)
    result += String.fromCharCode(text.charCodeAt(i) - 32)

  return result
}

describe('Upper case properties', () => {

  it('Resulting string doesn\'t contain lower case letters', () => {
    fc.expect(fc.scenario()
      .forall('text', fc.string(0, 10, fc.char('a', 'z')))
      .then(({text}) => !/[a-z]/.test(upperCase(text)))     //regex
      .check()
    )
  })

  it('Resulting string has the same length as original', () => {
    fc.expect(fc.scenario()
      .forall('text', fc.string(0, 10))
      .then(({text}) => text.length === upperCase(text).length)
```

```
      .check()
    )
  })

})
```

D.2 Task 2

D.2.1 Group A

```
import * as fc from '../../src/index'
import {it} from 'mocha'

function swap(items, leftIndex, rightIndex) {
  const temp = items[leftIndex]
  items[leftIndex] = items[rightIndex]
  items[rightIndex] = temp
}

function partition(items, left, right) {
  const pivot = items[Math.floor((right + left) / 2)]
  let i = left
  let j = right
  while (i <= j) {
    while (items[i] < pivot)
      i++
    while (items[j] > pivot)
      j--
    if (i <= j) {
      swap(items, i, j)
      i++
      j--
    }
  }
  return i
}

function quickSort(items, left, right) {
  if (items.length > 1) {
    const index = partition(items, left, right)
    if (left < index - 1)
      quickSort(items, left, index - 1)
    if (index < right - 1)
      quickSort(items, index, right - 1)
  }
  return items
```

```
}

function quick_sort(items) {
  return quickSort(items, 0, items.length - 1)
}

describe('QuickSort properties', () => {
  let seededGen: (seed: number) => () => number

  beforeEach(() =>
    seededGen = (seed: number) => () => (seed = seed * 16807 % 2147483647) /
        2147483647
  )

  it('Placeholder property', () => {
    fc.expect(fc.scenario()
      //.configStatistics(fc.statistics().withAll('quicksort_p1').withDefaultGraphs
          ())    //displays information after a run and also generates csv and
          graphs
      .withGenerator(seededGen)
      .forall('a', fc.integer(-10,10))
      .then(({a}) => a === a)
      .check()
    )
  })
})
```

D.2.2 Group B

```

import * as fc from '../../src/index'
import {it} from 'mocha'

function swap(items, leftIndex, rightIndex) {
  const temp = items[leftIndex]
  items[leftIndex] = items[rightIndex]
  items[rightIndex] = temp
}

function partition(items, left, right) {
  const pivot = items[Math.floor((right + left) / 2)]
  let i = left
  let j = right
  while (i <= j) {
    while (items[i] < pivot)
      i++
```

```
    while (items[j] > pivot)
      j--
    if (i <= j) {
      swap(items, i, j)
      i++
      j--
    }
  }
  return i
}

function quickSort(items, left, right) {
  if (items.length > 1) {
    const index = partition(items, left, right)
    if (left < index - 1)
      quickSort(items, left, index - 1)
    if (index < right - 1)
      quickSort(items, index, right - 1)
  }
  return items
}

function quick_sort(items) {
  return quickSort(items, 0, items.length - 1)
}

describe('QuickSort properties', () => {
  it('Placeholder property', () => {
    fc.expect(fc.scenario()
      .forall('a', fc.integer(-10,10))
      .then(({a}) => a === a)
      .check()
    )
  })
})
```

D.3 Example

```
import * as fc from '../../src/index'
import {it} from 'mocha'

// Code under test

const inc = (x: number): number => x + 1

const dec = (x: number): number => x - 1
```

```
9
10 const sum = (a: number, b: number): number => {
11   if (b > 0)
12     for (let i = 0; i < b; i++)
13       a = inc(a)
14   else
15     for (let i = 0; i < -b; i++)
16       a = dec(a)
17   return a
18 }
19
20 // Sum Properties
21 describe('sum', () => {
22   // Comutative property
23   it('Comutative property of addition', () => {
24
25     fc.expect(fc.scenario()
26       .forall('a', fc.integer(-100, 100))
27       .forall('b', fc.integer(-100, 100))
28       .then(({a, b}) => sum(a,b) === sum(b,a))
29       .check()
30     )
31
32   })
33
34   // Associative property
35   it('Associative property of addition', () => {
36
37     fc.expect(fc.scenario()
38       .config(fc.strategy().defaultStrategy().withSampleSize(21))
39       .forall('a', fc.integer(-100, 100))
40       .forall('b', fc.integer(-100, 100))
41       .forall('c', fc.integer(-100, 100))
42       .then(({a, b, c}) => sum(sum(a,b), c) === sum(a, sum(b,c)))
43       .check()
44     )
45
46   })
47
48   // Identity property
49   it('Identity property of addition', () => {
50
51     fc.expect(fc.scenario()
52       .config(fc.strategy().defaultStrategy().withSampleSize(10000))
53       .forall('a', fc.integer(-5000, 5000))
54       .then(({a}) => sum(a,0) === a)
55       .check()
56     )
57
```

```
58    })
59
60    // Closure property
61    it('Closure property of addition', () => {
62
63      fc.expect(fc.scenario()
64        .forall('a', fc.integer(-100, 100))
65        .forall('b', fc.integer(-100, 100))
66        .then(({a, b}) => Number.isInteger(sum(a,b)))
67        .check()
68      )
69
70    })
71
72    // Extra property
73    it('Extra property of addition', () => {
74
75      fc.expect(fc.scenario()
76        .forall('a', fc.integer(-100, 100))
77        .forall('b', fc.integer(-100, 100))
78        .then(({a, b}) => sum(a,b) === sum(a-1,b+1))
79        .check()
80      )
81
82    })
83
84  })
```

References

[1] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D³ data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, 2011.

[2] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[3] Henry Coles. Quicktheories. Available at https://github.com/quicktheories/QuickTheories, 2019. Accessed: 2021-02-03.

[4] Thomas D Cook, Donald Thomas Campbell, and William Shadish. *Experimental and quasi-experimental designs for generalized causal inference*. Houghton Mifflin Boston, MA, 2002.

[5] Joost CF De Winter. Using the student's t-test with extremely small sample sizes. *Practical Assessment, Research, and Evaluation*, 18(1):10, 2013.

[6] Nicolas Dubien. fast-check. Available at https://github.com/dubzzz/fast-check, 2020. Accessed: 2021-01-26.

[7] G. Fink, K. Levitt, M. Bishop, and M. Helmke. An interface language between specifications and testing. 1995.

[8] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. *SIGSOFT Softw. Eng. Notes*, 22(4):74–80, July 1997.

[9] OpenJS foundation. Node.js. Available at https://nodejs.org/en/, 2021. Accessed: 2021-06-03.

[10] HCl-10. Keys counter. Available at https://marketplace.visualstudio.com/items?itemName=HCl-10.vscode-keys-counter, 2020. Accessed: 2021-06-21.

[11] Fred Hebert. Proper testing - shrinking. Available at https://propertesting.com/book_shrinking.html, 2017. Accessed: 2021-02-03.

[12] Christopher Hiller. Mocha. Available at https://mochajs.org/, 2021. Accessed: 2021-06-03.

[13] Paul Holser. junit-quickcheck. Available at https://github.com/pholser/junit-quickcheck, 2021. Accessed: 2021-01-27.

[14] Shahin Honarvar, Mohammad Reza Mousavi, and Rajagopal Nagarajan. Property-based testing of quantum programs in q#. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, ICSEW'20, page 430–435, New York, NY, USA, 2020. Association for Computing Machinery.

[15] D. Kalpic, N. Hlupic, and Miodrag Lovric. Student's t-tests. In *International Encyclopedia of Statistical Science*, page 1559–1563, 2011.

[16] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Johannes Link. shrinking-challenge. Available at https://github.com/jlink/shrinking-challenge, 2021. Accessed: 2021-01-31.

[18] Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 46–56, New York, NY, USA, 2017. Association for Computing Machinery.

[19] David MacIver, Zac Hatfield-Dodds, and Many Contributors. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software*, 4:1891, 11 2019.

[20] David R. MacIver. Compositional shrinking. Available at https://hypothesis.works/articles/compositional-shrinking/, 2016. Accessed: 2021-02-03.

[21] David R. MacIver. Integrated vs type based shrinking. Available at https://hypothesis.works/articles/integrated-shrinking/, 2016. Accessed: 2021-01-31.

[22] Charles E Mackenzie. *Coded-Character Sets: History and Development*. Addison-Wesley Longman Publishing Co., Inc., 1980.

[23] Meryn. performance-now. Available at https://www.npmjs.com/package/performance-now, 2017. Accessed: 2021-06-03.

[24] Microsoft. Visual studio code. Available at https://code.visualstudio.com/, 2021. Accessed: 2021-06-21.

[25] Harvey Motulsky. Interpreting results: Mann-whitney test. Available at https://www.graphpad.com/guides/prism/latest/statistics/how_the_mann-whitney_test_works.htm, 2021. Accessed: 2021-06-26.

[26] Rohan Padhye. Jqf. Available at https://github.com/rohanpadhye/JQF, 2020. Accessed: 2021-01-29.

[27] Rohan Padhye, Caroline Lemieux, and Koushik Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 398–401, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA, 2019. Association for Computing Machinery.

[29] Manolis Papadakis and Konstantinos Sagonas. A proper integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, Erlang '11, page 39–50, New York, NY, USA, 2011. Association for Computing Machinery.

[30] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1410–1421, New York, NY, USA, 2020. Association for Computing Machinery.

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

[32] Hugo Sereno Ferreira, André Restivo, and Rui Gonçalves. Fluent check: A fluent property-based testing framework with existential quantification, 2021.

[33] Scott Vokes. theft. Available at https://github.com/silentbicycle/theft, 2019. Accessed: 2021-02-03.

[34] Scott Wlaschin. Choosing properties for property-based testing. Available at https://fsharpforfunandprofit.com/posts/property-based-testing-2/, 2014. Accessed: 2021-01-25.

[35] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[36] Michał Zalewski. american fuzzy lop. Available at https://lcamtuf.coredump.cx/afl/, 2014. Accessed: 2021-01-27.

[37] Q. Zhang, J. Wang, M. A. Gulzar, R. Padhye, and M. Kim. Bigfuzz: Efficient fuzz testing for data analytics using framework abstraction. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 722–733, Sep. 2020.