FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Towards a Support Tool for Prolog Using Property-Based Testing

Fábio Araújo de Sá

Master in Informatics and Computing Engineering

Supervisor: Prof. António Mário da Silva Marcos Florido

Supervisor: Prof. Daniel Augusto Gama de Castro Silva

Supervisor: Prof. Gonçalo da Mota Laranjeira Torres Leão

July 28, 2025

Towards a Support Tool for Prolog Using Property-Based Testing

Fábio Araújo de Sá

Master in Informatics and Computing Engineering

July 28, 2025

Resumo

Property-Based Testing (PBT) é uma técnica de teste de software que permite a geração automática de casos de teste com base em propriedades definidas, proporcionando uma cobertura mais abrangente do que os testes unitários tradicionais. No entanto, a aplicação desta metodologia em linguagens como Prolog levanta desafios específicos devido à sua natureza declarativa e não-determinística, bem como à ausência de um sistema de tipos explícito. Esta dissertação insere-se neste contexto, procurando explorar de que forma PBT pode ser eficientemente implementado em Prolog, com vista a melhorar tanto a qualidade do software como os processos de ensino e avaliação nesta linguagem de programação.

O trabalho desenvolvido incidiu na análise e adaptação de algoritmos para geração e redução de casos de teste, adequados ao paradigma lógico. Foi criada uma nova *framework* de PBT usando SICStus Prolog, tendo em consideração características como a unificação, o *backtracking* e a inexistência de tipos. A *framework* implementada inclui geradores de dados básicos e compostos, algoritmos de redução e apresentação de contra-exemplos, e funcionalidades de avaliação de propriedades com contagem de soluções. O sistema permite também a simulação de interações com a base de conhecimento e a análise de predicados com comportamento não-determinístico, estendendo a aplicabilidade do PBT a um conjunto alargado de cenários típicos de Prolog.

Foi posteriormente conduzido um estudo num sistema automático de avaliação académica, comparando a eficácia do PBT com a dos testes unitários tradicionais, com base em critérios como a detecção de erros, o tempo de execução e a complexidade do código de teste. O estudo realizado evidenciou que a abordagem baseada em propriedades pode detectar erros lógicos e omissões com maior profundidade do que os testes unitários tradicionais. Foram também identificadas melhorias significativas na clareza do feedback fornecido aos utilizadores, contribuindo para processos de aprendizagem mais eficazes e autónomos.

Esta dissertação também abre portas a investigações futuras, nomeadamente à integração desta *framework* em plataformas de ensino, à avaliação de predicados que requerem interacção com o utilizador através da consola, ou à geração e teste de bases de factos com estruturas mais complexas, como factos interdependentes ou com relações lógicas entre si.

Keywords: Logic Programming, Prolog, Testing, Property-Based Testing

Abstract

Property-Based Testing (PBT) is a software testing technique that enables the automatic generation of test cases based on defined properties, providing broader coverage than traditional unit tests. However, applying this methodology in languages such as Prolog raises specific challenges due to its declarative and non-deterministic nature, as well as the absence of an explicit type system. This dissertation fits into this context, aiming to explore how PBT can be efficiently implemented in Prolog, with the goal of improving both software quality and the teaching and assessment processes in this programming language.

The work developed focused on the analysis and adaptation of algorithms for test case generation and shrinking, suitable for the logic programming paradigm. A new framework for PBT was created using SICStus Prolog, taking into account features such as unification, backtracking, and the lack of types. The implemented framework includes generators for basic and composite data, shrinking algorithms and counterexample reporting, as well as functionalities for property evaluation with solution counting. The system also allows the simulation of interactions with the knowledge base and the analysis of predicates with non-deterministic behavior, extending the applicability of PBT to a wide range of typical Prolog scenarios.

A study was subsequently conducted in an automatic academic assessment system, comparing the effectiveness of PBT with that of traditional unit testing, based on criteria such as error detection, execution time, and test code complexity. The study showed that the property-based approach can detect logical errors and omissions more deeply than traditional unit tests. Significant improvements were also identified in the clarity of the feedback provided to users, contributing to more effective and autonomous learning processes.

This dissertation also opens the door to future research, namely the integration of this framework into educational platforms, the evaluation of predicates that require user interaction through the console, or the generation and testing of fact bases with more complex structures, such as interdependent facts or logical relationships between them.

Keywords: Logic Programming, Prolog, Testing, Property-Based Testing

Acknowledgements

Six years ago, I didn't know which area to choose. Five years ago, I nervously enrolled in the Informatics and Computing course. Two years ago, I decided to pursue a Master's degree, even though I was already working, and a year ago I received and accepted this thesis proposal as a way to complete my academic journey in the best possible way. Now, I feel that every decision was worth it, and that my entire journey has been guided by fantastic people who deserve to be acknowledged on this page.

To my parents, thank you for all the support from the very beginning of my academic life, which back in 2020 we weren't sure I would be able to complete for various reasons.

To my classmates who became true friends, thank you for understanding when I couldn't join every outing, for the countless sleepless nights, and for the many hours of shared study.

To my supervisors, Professors Mário Florido, Daniel Silva, and Gonçalo Leão, thank you for all the support shown, meeting after meeting throughout the past year, for your ability to share knowledge so openly, and for your endless understanding of my time constraints.

To all the fellow students I had the pleasure of crossing paths with during this journey, my sincere thanks.

Fábio Araújo de Sá

"Os únicos interessados em mudar o mundo são os pessimistas,
porque os otimistas estão encantados com o que há"

José Saramago

Contents

List of Figures

List of Algorithms

List of Tables

List of Listings

Abbreviations

AI	Artificial Intelligence
ADT	Abstract Data Type
API	Application Programming Interface
CLP	Constraint and Logic Programming
FEUP	Faculty of Engineering of the University of Porto
HC	Hill Climbing
ISO	International Organization for Standardization
JITI	Just-In-Time Indexing
ML	Machine Learning
NF	Neighboring Function
OOP	Object-Oriented Programming
PFL	Functional and Logical Programming
PBT	Property-Based Testing
PRNG	Pseudo-Random Number Generator
SA	Simulated Annealing
SS	Search Strategy
SUT	System Under Test
TG	Targeted Generator
TPBT	Targeted Property-Based Testing
UT	Unit Testing
UV	Utility Value
WAM	Warren Abstract Machine
YAP	Yet Another Prolog

Chapter 1

Introduction

Software systems are becoming increasingly commonplace, forming the foundation for countless processes and applications that power contemporary technology, making it increasingly difficult to guarantee their quality and dependability as they grow in complexity. This chapter introduces the research context of this dissertation, together with the motivations behind it and the objectives to be achieved. It also presents the problems identified and the questions that guided the development of this work.

1.1 Context

Software testing plays an important role in the development and validation of computer systems, ensuring the quality and reliability of the final product [Quadri and Farooq, 2010, Bissi et al., 2016]. Although Unit Testing (UT) is widely recognized in both industrial and academic settings, where specific and predefined test cases are used, it becomes less effective when dealing with systems that involve a vast range of possible inputs, which are impractical to cover manually [Jamil et al., 2016]. This challenge calls for alternative approaches better suited to addressing such complexity: Property-Based Testing (PBT).

PBT was initially introduced and popularized by QuickCheck [Claessen and Hughes, 2000] for Haskell programming language. Since its implementation and release two decades ago, PBT has gained traction in the market, and several implementations have emerged for different programming languages, such as Hypothesis[1] in Python, PropEr[2] in Erlang or PrologCheck [Amaral et al., 2014] in Prolog.

PBT offers advantages over traditional software testing methods, as it allows programmers to define formal but more programmer-friendly rules that the system must follow. This method automatically generates a variety of input scenarios and tests the system with a much wider range of possibilities than any manual implementation [Ziat et al., 2021]. As a result, there is greater

[1] Available online at: `https://hypothesis.readthedocs.io/en/latest/` (last access on June, 2025)
[2] Available online at: `https://github.com/proper-testing/proper` (last access on June, 2025)

testing coverage and also the exploration of situations that might not have been contemplated with other methods, thus gaining a robust layer of protection.

Another characteristic of the property-based approach is the automatic shrinking process. When a test fails, there is a systematic reduction of the corresponding input, simplifying it to its most basic form and identifying minimal, reproducible test cases [Lo et al., 2020]. This ensures that the failure case analysis is also simplified, and the cause is identified more quickly, reducing debugging time and enabling continuous improvement in the accuracy of the software produced.

1.2 Motivation

Property-Based Testing is a powerful approach that can benefit any programming language or system, as the core techniques are language-agnostic. This abstraction enables the expression of program properties in a more high-level, readable, and maintainable way compared to traditional unit tests. As a result, testers can focus more directly on the system's requirements and expected behavior [Goldstein et al., 2024], which improves the clarity and scope of the testing process.

Moreover, the application of PBT in educational environments, where Prolog is commonly used to teach logic programming, could significantly enhance the quality of the feedback and the assessment [Benac Earle et al., 2016]. The automatic generation of diverse test cases and counterexamples could save time and effort, allowing educators and professors to focus more on helping students understand the core concepts of the language rather than spending time on test creation and manual assessment.

In addition to its technical contributions, this work aligns with United Nations Sustainable Development Goals[3]. Specifically, it relates to SDG 4 by encouraging inclusive and equitable quality education through the integration of Property-Based Testing into logic programming courses using Prolog. This approach can support improvements in feedback quality and consistency, contributing to better learning experiences. It also aligns with SDG 9 by exploring innovative testing techniques and expanding the availability of software infrastructure in educational and development contexts. These aspects reinforce the relevance and potential impact of the proposed work.

1.3 Problem Definition

Prolog is a declarative programming language that presents unique challenges for the implementation of Property-Based Testing. Its non-deterministic nature and the lack of an explicit type system make the generation of instances and the shrinking process more complex than for strongly typed languages [MacIver and Donaldson, 2020].

One of the main issues in testing Prolog systems is the lack of detailed failure information when a test does not pass. In contrast to languages with built-in exception handling and structured error messages, like Python or Java, or strongly typed languages, like Haskell or Rust, Prolog frequently just indicates that a goal has failed without providing an explanation of why, which

[3] Available online at: `https://sdgs.un.org/goals` (last access on June, 2025)

clause was at fault, or what aspect of the input caused the failure [Cohen, 1985]. It is challenging to identify the underlying cause of a testing error due to this lack of feedback. However, by generating input and using shrinking in PBT to produce meaningful counterexamples, it becomes possible to gain clearer insights into the failing cases, reducing the time spent interpreting failure messages and enhancing the debugging process.

Furthermore, Prolog does not have a well-maintained library that supports all the key features commonly used in Property-Based Testing. Existing tools like PrologCheck [Amaral et al., 2014] and QuickCheck for Prolog[4] attempt to address some of these issues but still leave much to be desired, particularly in terms of instance generation and shrinking, which could be made more efficient with the use of algorithms found in other libraries.

The development of a more robust PBT framework for Prolog would address these issues, providing better support for immediate feedback and making the testing process more efficient and effective in both academic and professional environments.

1.4 Objectives

This research addresses these gaps by developing a more robust PBT framework for Prolog, contributing both to the improvement of software quality and to the enhancement of educational experiences in programming courses. It also explores the best approach for implementing PBT in Prolog efficiently, while adhering to programming language best practices. The research presented throughout this document addresses the following research questions:

- **RQ1:** Which algorithms for instance generation and shrinking are most suitable for Property-Based Testing in Prolog?
- **RQ2:** How can the characteristics of logic programming and non-deterministic languages like Prolog be exploited in these algorithms?
- **RQ3:** What is the impact of Property-Based Testing in Prolog, particularly in academic automatic grading systems, and how does it compare to traditional unit tests in this context?

To answer these research questions, the dissertation adopts a combination of theoretical evaluation and practical experimentation. For the first two research questions, RQ1 and RQ2, a theoretical review of the most commonly used algorithms in Property-Based Testing is conducted. This includes an analysis of existing tools such as QuickCheck, Hypothesis, and Falsify, to identify the most suitable algorithms for generating instances and performing shrinking processes in the context of Prolog, considering specific Prolog features, including unification, backtracking, and its depth-first nature. The selection process also accounts for the flexibility needed to override the behavior of the generators and shrinkers for testing specialized data structures.

[4] Available online at: `https://github.com/nicoabie/quickcheck` (last access on June, 2025)

For the third research question, RQ3, a case study is conducted in the context of automatic grading systems. The impact of PBT on these systems is assessed by comparing it to traditional UT in terms of fault detection efficiency, execution time, and the complexity of test code.

The methodology outlined in Chapter 4 guides the implementation and evaluation of the proposed algorithms and case studies, providing a comprehensive analysis of their performance and practical application.

1.5 Contributions

This dissertation makes contributions to the field of software testing in the context of logic programming, introducing adaptations of Property-Based Testing techniques to Prolog that address limitations in instance generation and shrinking, with the following outcomes:

- Cataloging comprehensively the main existing PBT libraries, relating them to the programming languages developed, as well as associating them with the programming paradigm used, whether the language is typed or untyped, compiled or interpreted;
- Gathering state-of-the-art algorithms for PBT, both in input generation and shrinking processes, understanding the main approaches of these algorithms, their advantages and disadvantages, and mapping them directly to the properties of the languages used;
- Identification of the most suitable algorithms for Property-Based Testing in Prolog, specifically tailored to leverage the unique features of logic programming and non-deterministic languages;
- Development and implementation of these algorithms in Prolog, providing a framework for applying Property-Based Testing in this context;
- Evaluation of the impact of Property-Based Testing on automatic grading systems, demonstrating how it compares to traditional unit testing approaches in terms of fault detection, performance, and ease of use;
- Establishes a foundation for future extensions of Property-Based Testing in Prolog, particularly in testing predicates involving input and output and complex, interrelated knowledge bases, highlighting the feasibility of simulating I/O streams and leveraging Constraint Logic Programming (CLP) to generate semantically coherent test data.

1.6 Document Structure

This document is divided into six additional chapters. Chapter 2 provides the contextualization, focusing on software testing and programming language features. It highlights the key concepts and their relevance to the challenges addressed in this research. Chapter 3 presents the state-of-the-art PBT tools, libraries, and algorithms for input generation and shrinking, discussing their

limitations and identifying gaps that motivate this work, as well as PBT usage in academic fields. Chapter 4 outlines the proposed methodology and evaluation plan, detailing the approach, experimental design, potential risks, and mitigation strategies. Chapter 5 presents the theoretical selection of algorithms, the design and development of the PBT library, and an explicit description of all implemented features and known issues. Chapter 6 evaluates the framework in the context of automatic correction systems, highlighting both the strengths and the limitations of the implementation. Finally, Chapter 7 summarizes the main contributions and potential future work.

Chapter 2

Fundamentals of Software Testing and Programming Paradigms

This chapter introduces essential concepts in software testing and programming language paradigms that form the theoretical foundation for the application of Property-Based Testing. Firstly, the main concepts and types of software testing are discussed, highlighting their importance for quality assurance and the differences inherent in the different techniques. Next, the taxonomy and paradigms of programming languages are explored, focusing on their characteristics and the support offered, with particular emphasis on Prolog, as it is the language adopted for the implementation.

2.1 Software Testing

Software testing is a fundamental practice in the software development lifecycle, playing an essential role in quality assurance and risk mitigation [Quadri and Farooq, 2010]. The concept of software testing dates back to the early years of programming, when the growing complexity of systems began to reveal recurring faults that compromised their operation [Myers, 1979]. Testing, initially an ad hoc activity, evolved in parallel with the development of software engineering, becoming treated as a formal discipline.

In recent years, with the increased criticality and complexity of software systems, the field of software testing has expanded. Software testing is vital for ensuring that a system fulfills the established functional and non-functional requirements [Jamil et al., 2016]. Its role is to identify defects in the System Under Test (SUT), prevent catastrophic failures, and ensure that the system operates within acceptable quality and performance standards. When well implemented, testing significantly reduces the risk of defects only being discovered in production, where the costs of correction are exponentially higher, ensuring that the final product delivered to the customer is in line with expectations, more robust, and secure [Bissi et al., 2016].

However, although testing is essential, it does not guarantee the absence of errors. One of the most important principles of testing practice is that it reveals the presence of defects, but can never prove that the system is completely fault-free [Geruslu et al., 2020]. In non-trivial software

systems, the number of combinations of inputs and states is so vast that testing all permutations would be unfeasible, both in terms of time and cost. Even with a substantial testing effort, there is also the risk that inadequate test planning or inefficient execution can result in insufficient test coverage. Test coverage is a metric that indicates the proportion of a software's source code or requirements that have been executed or verified by a set of tests, helping to evaluate the effectiveness of the testing strategy [Quadri and Farooq, 2010]. Maintaining tests over time can be costly, especially if the software undergoes many changes, requiring test cases to be constantly adjusted.

Software testing has become an essential component of academic curricula in software engineering and computer science programs worldwide. As software systems grow increasingly complex, universities recognize the need to equip students with robust testing skills to meet industry demands [Geruslu et al., 2020]. Teaching software testing poses several challenges, such as developing effective pedagogical strategies and providing hands-on experience with real-world and specialized tools, case studies, and project-based learning.

There are various types of software testing, each with a specific objective and scope, depending on the context of the project and the type of application being developed [Umar, 2020]. These can be broadly categorized into two families: Closed-box Testing and Open-box Testing, with several subtypes under each. For instance, Open-box Testing encompasses techniques such as unit testing, integration testing, path testing, and branch testing, among others, while Closed-box Testing includes approaches like functional testing, boundary testing, exploratory testing, and property-based testing, as shown in Figure 2.1.

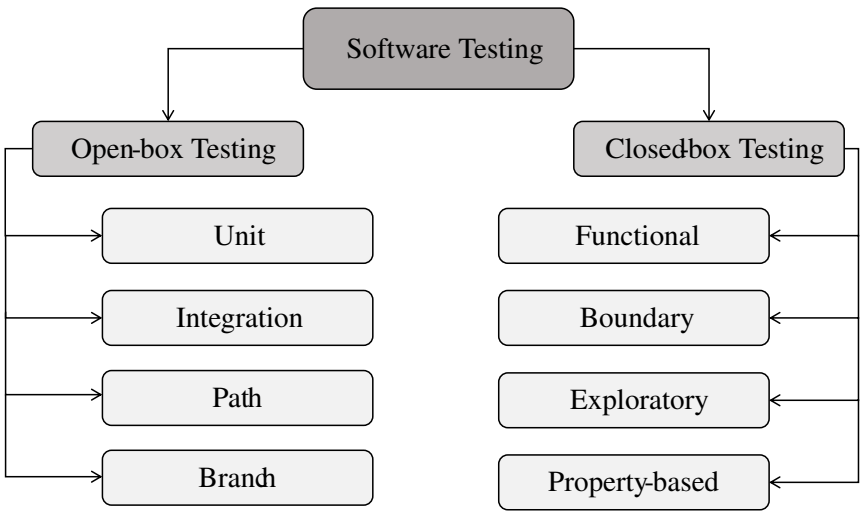

Figure 2.1: Hierarchy of Testing Types. Adapted from [Umar, 2020].

Open-box testing involves examining the internal workings of the software, including unit testing (verifying individual components), integration testing (checking interactions between components), path testing (ensuring all execution paths are covered), and branch testing (validating all decision points). In contrast, closed-box testing focuses on the software's external behavior, including functional testing (checking features against requirements), boundary testing (evaluating

edge cases), exploratory testing (freely probing the system for unexpected issues), and property-based testing (verifying general rules hold under various inputs).

This dissertation, however, focuses on two specific testing methodologies: Unit Testing and Property-Based Testing. These testing approaches are chosen because they represent a broad spectrum of techniques that can be applied across various stages of the software development lifecycle. They range from functional verification to deeper explorations of system behaviors using automated test generation. While Unit Testing is among the most well-established type of software testing, Property-Based Testing is a more recent technique that has gained increasing attention for its ability to automatically generate test cases to explore edge cases and uncover defects that might be difficult to identify with conventional testing methods [Lo et al., 2020]. The discussion begins with an overview of Open-box Testing, followed by a deeper examination of Unit Testing. Subsequently, Close-box Testing techniques are addressed, culminating in a detailed analysis of PBT, which receives particular emphasis in this dissertation due to its innovative approach and increasing significance in modern software development.

2.1.1 Open-box Testing

Open-box is a testing technique that requires explicit knowledge about the internal workings of the software in order to select test data [Verma et al., 2017]. It evaluates the program's functions and methods and is most effective when the tester has a clear understanding of what the software is supposed to do, as shown in Fig. 2.2.

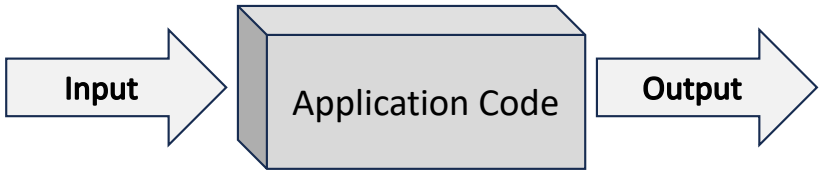

Figure 2.2: Open-box Testing. Adapted from [Verma et al., 2017].

The advantages of open-box testing include introspection, which allows testers to identify all of the software's functions and thus ensure wider coverage of potential test cases. The approach also enables a thorough exploration of all internal paths and subsequent interactions, leading to the identification of hidden bugs. Complexity is one factor to consider, as the tester needs to have detailed knowledge of how the application works, which can require a significant investment in time and effort [Geruslu et al., 2020].

2.1.2 Unit Testing

Unit testing is a software testing technique that consists of checking individual components or units of code, such as functions or methods, to ensure that each part works correctly in isolation

[Klammer and Kern, 2015]. This approach is fundamental in the development phase, as it allows developers to identify and correct problems in specific units before they spread to other parts of the system.

Unit tests aim to validate the behavior of small parts of the code, making it easier to detect defects early on. The central principle is that each unit should be tested in isolation, ensuring that the implemented logic works as expected. To do this, test frameworks are used that enable the automation and efficient execution of tests, such as JUnit[1] for Java or pytest[2] for Python. The code displayed in Listing 2.1 serves as an illustration of unit testing with the pytest framework. It includes multiple test cases intended to verify the functionality of a `reverse` function. A variety of situations are covered by these tests, such as reversing non-empty strings, empty strings, single characters, and strings with different lengths.

```
import pytest

def test_reverse_simple():
    assert reverse('hello') == 'olleh'

def test_reverse_empty_string():
    assert reverse('') == ''

def test_reverse_single_character():
    assert reverse('a') == 'a'

def test_reverse_2_characters():
    assert reverse('ab') == 'ba'

def test_reverse_3_characters():
    assert reverse('abc') == 'cba'
```

Listing 2.1: Unit testing using pytest framework

One of the main advantages of unit testing is the early detection of errors. Validating units in the early stages of development significantly reduces the cost and effort required to correct defects in later stages of the software lifecycle [Quadri and Farooq, 2010], provides automatic documentation of the expected behavior of the units, making it easier to maintain the code when making modifications or updates. These tests are simpler to write, allowing less experienced developers to contribute effectively.

However, unit testing coverage is often limited, since although tests are effective at verifying the internal workings of units, they do not guarantee that the integration between different components works correctly [Klammer and Kern, 2015]. They are sensitive to changes in code implementation, which can require updating many tests and increasing the maintenance overhead.

[1] Available online at: https://junit.org/junit5/ (last access on June, 2025)
[2] Available online at: https://docs.pytest.org/en/stable/ (last access on June, 2025)

Finally, because tests focus on isolated units, they may not capture problems related to interactions between different system parts.

Unit testing is used as a comparator in this dissertation because of its wide acceptance in industry and academic areas and its effectiveness in detecting problems in the early stages of development [Bissi et al., 2016]. This facilitates a comprehensive analysis of how Property-Based Testing complements or diverges from conventional practices, offering insights into its potential benefits and adoption barriers across different domains.

2.1.3 Closed-box Testing

Closed-box testing is a software testing technique in which the tester has no knowledge of the software's internal methods or structures [Quadri and Farooq, 2010]. In this approach, the tester only knows the system's inputs and what is expected as outputs, relying exclusively on the program's requirements, as shown in Fig. 2.3.

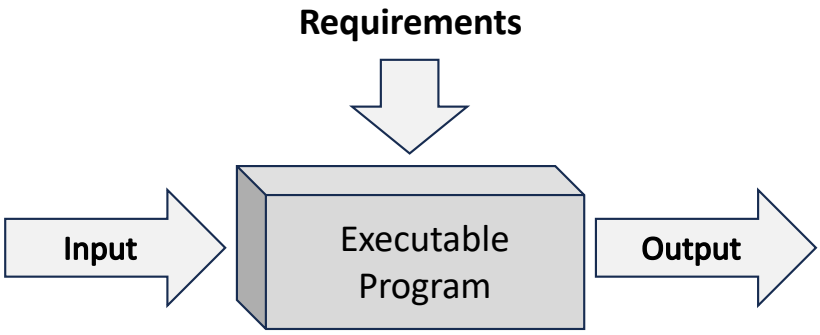

Figure 2.3: Closed-box Testing. Adapted from [Verma et al., 2017].

One of the main advantages of closed-box testing is that it results in unbiased testing [Umar, 2020]. This perspective allows testing to be carried out from the end user's point of view, which is crucial to ensuring that the software meets expectations and requirements. As the developers and testers are usually different members of the team, this separation minimizes bias and ensures that the test is not influenced by insider knowledge of the system. Another positive point is the fast development of test cases since testers can draw up these cases as soon as the specifications are complete, without the need to understand the internal paths of the code.

However, the test coverage can be incomplete, since testing all possible input paths is an impractical or excessively time-consuming task, resulting in many program paths remaining untested [Verma et al., 2017]. Maintenance is also a challenge, as constant changes to the specification may require testers to frequently rewrite tests, [Quadri and Farooq, 2010], increasing the time and effort required to maintain the test suite, which is an organized set of test cases that are run together to validate the functionality and behavior of a piece of software against its specifications.

2.1.4 Property-Based Testing

Property-Based Testing is a methodology that focuses on validating a system's behavior by testing general properties, or invariants, rather than specific input-output pairs [Chen et al., 2022]. Unlike traditional Unit Testing, which assesses functionality using predefined examples, PBT defines properties that must consistently hold true across a wide range of inputs.

PBT is particularly well-suited to Closed-box Testing, where the emphasis lies on observable behaviors rather than internal implementation details [Goldstein et al., 2024]. This testing method allows for comprehensive testing without necessitating an understanding of the inner workings of the system by using characteristics to specify how the system should react to different types of inputs. This method probes the limits of the system and reveals hidden edge cases by using random or generative approaches to generate a wide range of input possibilities. Because it ensures consistent and dependable performance under a range of input situations, PBT is a flexible and important tool for testing systems with intricate or opaque implementations.

Property-Based Testing has its roots in the theoretical developments of the 1970s and 1980s, when new paradigms for software validation were being shaped by formal verification and logic programming [Fink and Bishop, 1997]. During this time, methods based on generic principles or characteristics began to take precedence over example-based testing. The methods that would eventually define PBT were founded on the recognition by researchers that checking a system against overarching invariants may offer a more thorough and trustworthy foundation for guaranteeing correctness.

The development of QuickCheck by Claessen and Hughes [Claessen and Hughes, 2000] for the Haskell programming language marked a turning point in the development of PBT. It introduced the idea of specifying properties as first-class entities in testing and allowed programmers to use formal statements rather than discrete instances to capture the intended behavior of programs. This framework promoted a deeper comprehension of system behavior by making the testing process more abstract and systematic.

Unlike the example in Listing 2.1, where discrete test cases are specified explicitly, in Listing 2.2 the properties shown validate that reversing a single-element list preserves the element, concatenating and reversing sublists produces consistent results, and double reversal restores the original list.

```
1 property_rev_unit = reverse [x] == [x]
2
3 property_rev_sublist xs ys =
4     reverse (xs++ys) == reverse ys ++ reverse xs
5
6 property_rev_reverse xs =
7     reverse (reverse xs) == xs
```

Listing 2.2: Property definition using QuickCheck framework

A key component of property-based testing is the specification of properties, which act as logical predicates outlining the intended behavior of a system [Hughes, 2011]. These properties are used to verify that a system functions as intended and they represent invariants such as commutativity, idempotence, associativity, or reversibility. Focusing on generic system behaviors, PBT supports developers to adopt a grasp of requirements [Corgozinho et al., 2023, Lampropoulos et al., 2019], leading to more robust and maintainable designs.

In Listing 2.3, the property `property_rev_reverse`, previously defined in Listing 2.2, successfully passes 100 randomized tests when validated.

```
Main> quickCheck property_rev_reverse
OK passed 100 tests
```

Listing 2.3: Example of a successful QuickCheck test

The use of generators to automatically provide a wide range of input situations for testing characteristics is a crucial part of PBT [Ziat et al., 2021]. Compared to conventional example-based testing, these generators offer a far wider variety of inputs, probing the system with edge cases and surprising combinations that could otherwise go unnoticed. In addition to improving test coverage, this automatic input creation makes sure the system is evaluated against a wide range of real-world circumstances, boosting confidence in its robustness and dependability.

When a property fails, the shrinking process systematically reduces the complex failing input to its simplest form [Lo et al., 2020], making it easier to analyze and debug. Shrinking reduces the amount of time and effort required for debugging by reducing the failure to a minimal reproducible scenario, which enables developers to identify the underlying causes more quickly. As shown in Listing 2.4, the QuickCheck framework identifies a failure in the `property_rev_sublist` property after two tests. The shrinking mechanism reduces the inputs to `[2]` and `[-2, 1]`, which represent the simplest counterexample for this property.

```
Main> quickCheck property_rev_sublist
Falsifiable, after 2 tests:
[2]
[-2, 1]
```

Listing 2.4: Example of a failing QuickCheck test with shrinking

Despite all of its advantages, Property-Based Testing has drawbacks that, if not handled properly, could reduce its efficacy [Corgozinho et al., 2023]. The intricacy of defining significant properties is one of the main obstacles. Building characteristics that precisely represent the system's essential needs requires in-depth domain expertise and a thorough understanding of system behavior. Badly defined attributes have the potential to misrepresent the intended behavior or leave crucial functionality untested, which lowers the efficacy of the testing procedure. In addition, PBT relies on the completeness and accuracy of these qualities; any ambiguities or holes

in their formulation may result in overlooked edge cases or undiscovered flaws, particularly in complex systems.

The processing overhead involved in creating and analyzing big input sets is another significant disadvantage [MacIver and Donaldson, 2020]. Although PBT is excellent at exploring large input spaces, this benefit may become problematic in resource-intensive applications where thorough testing may not be feasible due to time and performance restrictions. Furthermore, a false sense of trust in the system's accuracy may result from reliance on particular properties [Corgozinho et al., 2023]. These features remain unvalidated and may result in undetected problems if the defined attributes fail to capture implicit or subtle needs.

Significant progress has been made in Property-Based Testing in recent years, especially in the fields of input creation and property identification. Instead of depending solely on random input creation, Targeted Property-Based Testing (TPBT) employs search algorithms to guide the input generation process, offering a substantial improvement over traditional PBT. Large input areas are frequently explored inefficiently in traditional PBT since the test inputs are produced at random and checked for attributes. By employing search techniques like simulated annealing or hill climbing, on the other hand, TPBT increases the possibility of finding counterexamples that contradict the tested qualities by methodically exploring the input domain [Löscher and Sagonas, 2017]. In situations where property violations are uncommon or there is a need for intricate input configurations that are unlikely to be generated at random, this approach seeks to increase the testing process' efficiency. Additionally, TPBT incorporates the use of utility values which measure the proximity of an input to violating the property under test. These values provide a feedback mechanism that informs the search strategy, allowing it to focus on inputs with higher potential for falsifying the property [Löscher and Sagonas, 2018].

According to the recent study carried out by Goldstein et al. [Goldstein et al., 2024], testers believe that property-based tests should be fast, performing on par with other unit tests, and used opportunistically when properties are readily available. On the other hand, testers typically don't look into characteristics that don't find bugs, even though they admit they should, and they prefer to use derived generators over new ones.

Property-Based Testing can also be applied to ML models to generate test inputs. This approach involves specifying desired properties for the model's behavior, such as fairness, robustness, or consistency, and systematically creating test cases that aim to violate these properties. Rather than relying on user-supplied generator functions, the ML model itself can be leveraged to generate the input data for testing. Sharma et al. [Sharma et al., 2021] proposed the open-box model training, such as a decision tree or a neural network, to approximate the behavior of the model under test. This open-box model could then be used to explore the input space and generate test data, aiming to find contradictions or failures in the specified properties. Using the model to generate inputs allows for a more dynamic and adaptive approach, improving test effectiveness without additional complexity in manually creating input properties or examples.

2.2 Languages and Paradigms

Programming languages emerged as a response to the need to interact with computers more efficiently and understandably than programming in machine language, which is complex and prone to errors [Floyd, 1979]. These early languages laid the foundation for distinct programming paradigms, each emphasizing different approaches to problem-solving.

2.2.1 Programming Languages

Programming languages are defined by their syntax, grammar, and semantics. Syntax and grammar govern how code is written and structured, while semantics define the meaning of each command. The programming languages are also classified by operational characteristics such as being interpreted or compiled, strongly or weakly typed, and static or dynamic [Samuel, 2018, Alves et al., 2023].

Compiled or Interpreted Compiled languages, such as C and C++, are transformed into machine code before execution, which generally results in superior performance. The compiler checks for syntax errors and generates an executable that can be run directly by the operating system [Janicic and Tošić, 2008]. In contrast, interpreted languages such as Python and JavaScript are executed line by line by an interpreter, which translates the code in real-time. This can make development and debugging easier, but often results in lower performance compared to compiled languages. On the other hand, Prolog can be used as either a compiled or an interpreted programming language [Cohen, 1985].

Strongly or Weakly typed Data types must be explicitly stated in strongly typed languages like Java and C++ to ensure code safety and resilience [Alves et al., 2023]. This means that if operations involving incompatible types are attempted, the compiler or interpreter will produce errors. In contrast, weakly typed languages like Python, JavaScript, and Prolog allow variables to be bound to different types at runtime, which provides flexibility. While this can facilitate development, it also introduces the possibility of errors that may be harder to detect.

Static or Dynamic Static typing languages, such as Java and C#, check data types at compile time, which means that developers need to specify types before execution. This offers advantages in terms of performance and security, but can make the code less flexible. In contrast, dynamically typed languages such as Ruby, Prolog, and Python allow types to be determined at runtime, facilitating prototyping and rapid iteration. This flexibility, however, can result in performance problems and a greater propensity for errors [Samuel, 2018].

Polymorphism According to Cardelli and Wegner [Cardelli and Wegner, 1985], polymorphism is the capacity of functions, operators, or entities to work with various data types or carry out various operations depending on their inputs. Depending on the kinds or quantity of arguments

it receives, the same function or operator can act differently according to this idea. For instance, when given integer inputs, a generic *sum* function may add integers. Yet, when given texts or lists, it may concatenate sequences. Greater abstraction and flexibility are made possible by polymorphism, which allows functions to be created to handle different kinds of structures, resulting in more flexible and maintainable code. Polymorphism is widely supported in languages like Python, Java, and JavaScript, enabling operators and functions to behave differently depending on the type of argument. Prolog's declarative paradigm, on the other hand, emphasizes logical inference over type-based function behavior, which limits its support for polymorphism [Cohen, 1985].

2.2.2 Programming Paradigms

A paradigm is a set of practices and concepts that guide the way programmers solve problems and structure their code [Samuel, 2018]. The fact that there is no one solution that works for all computing issues has contributed to the diversity of paradigms. This allows developers to select the best technique for each scenario, enhancing the code's expressiveness and maintainability.

The two primary kinds of programming paradigms are declarative and imperative, as shown in Fig. 2.4. Procedural and object-oriented imperative paradigms describe how a program should perform tasks by means of a series of instructions that alter the program's state. Declarative paradigms, such as functional and logic programming, place more emphasis on outlining the goals of the program than on how to get there. As a result, they frequently use less state and explicit control flow.

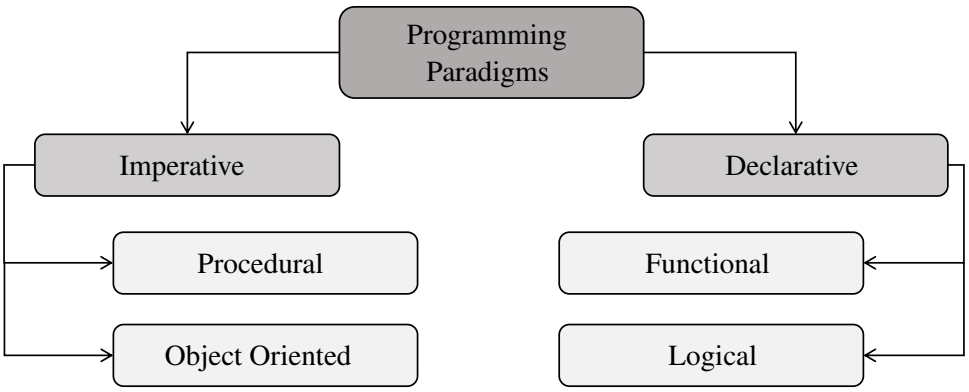

Figure 2.4: Hierarchy of Programming Language Paradigms. Adapted from [Samuel, 2018].

Procedural paradigm The procedural paradigm is a subset of the imperative paradigm and is centered on breaking down a program into reusable routines or procedures [Alves et al., 2023]. This approach organizes code into sequences of instructions, often in the form of functions or procedures, that operate on data and modify the program's state. Languages such as C and Pascal are classic examples of procedural programming, where control flow structures such as loops and conditionals play a central role in defining program behavior.

Object-oriented paradigm The Object-Oriented Paradigm (OOP) extends imperative programming by structuring code around *objects*, which combine data and behaviors. OOP allows developers to create modular and reusable code through encapsulation, inheritance, and polymorphism, making it well-suited for complex applications with interconnected components [Bartonícek, 2014]. Languages like Java and C# popularized this paradigm, offering a framework where data and methods are encapsulated within objects that interact with each other.

Functional paradigm The Functional paradigm is a declarative approach that treats computation as the evaluation of mathematical functions, avoiding mutable state and side effects. Functional programming encourages the use of pure functions, higher-order functions, and recursion, enabling a high degree of abstraction and composability [Samuel, 2018]. Languages like Haskell and Erlang are based on the functional paradigm, promoting a style of programming that is both expressive and well-suited for parallel processing.

Logic paradigm The logic paradigm focuses on defining relationships and rules, leaving the task of finding solutions to the underlying inference engine. This paradigm is grounded in formal logic, where programs consist of facts and rules that describe relationships between entities. Prolog is a primary example of logic programming [Warren et al., 2023], used frequently in areas like Artificial Intelligence (AI) and Natural Language Processing (NLP) due to its ability to model complex relationships and perform automatic reasoning.

Although many programming languages are often linked to a single paradigm, a number of modern languages support several paradigms, enabling programmers to integrate many approaches into a single program. Languages like Python, Scala, and JavaScript, for instance, are multi-paradigm and allow for procedural, object-oriented, and functional styles, providing the freedom to select the best paradigm for a particular problem.

2.2.3 Prolog

Introduced in the early 1970s, Prolog (short for Programming in Logic) provides a useful programming language based on first-order predicate calculus, operationalizing the Logic paradigm. The capacity of Prolog to specify relationships and enable a reasoning engine to draw conclusions is its fundamental feature [Cohen, 1985].

The foundation of Prolog's logical inference system is unification [Robinson, 1965], which allows terms to be compared and matched by identifying identical substitutions. In order to attain equality, variables are bound to terms by substitution in equations between terms, which must be solved. Recursively, the unification method compares constants for equality, ensures that compound terms have the same functors and arity, and binds variables to terms. This process plays a central role in Prolog's evaluation mechanism, driving the pattern-matching capabilities that are crucial for logic programming and problem-solving.

```
1 append([], Ys, Ys).
```

```
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).

?- append([a, b], [c, d], Result).
Result = [a, b, c, d] ?
```

Listing 2.5: Appending Lists in Prolog using Unification

The query presented in Listing 2.5 attempts to unify `append([a, b], [c, d], Result)` with the clauses of the `append/3` predicate. Through unification, Prolog binds variables step-by-step: `[a, b]` is unified with `[X|Xs]`, resulting in `X = a` and `Xs = [b]`; `[c, d]` is unified with `Ys`, resulting in `Ys = [c, d]`; and finally, the recursive calls resolve `Result` to `[a, b, c, d]`.

Backtracking, the fundamental computational technique of Prolog, methodically investigates potential solutions to a given problem [Apt, 1996]. It uses a Depth-First Search strategy [Tarjan, 1971] to traverse the search tree, which represents the logical derivation of answers, as illustrated in Fig. 2.5.

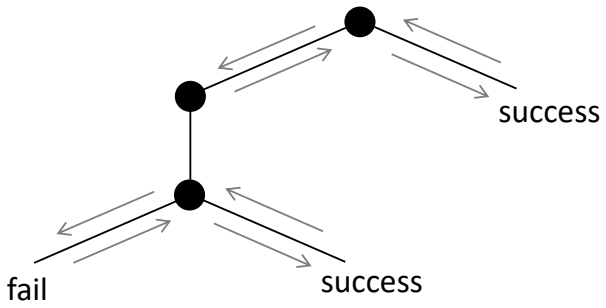

Figure 2.5: Backtracking Search Strategy. Adapted from [Apt, 1996].

The algorithm begins with the root query and explores one branch of the tree at a time. If a branch leads to failure, the search backtracks to the most recent decision point and explores alternative branches.

This method ensures systematic exploration of all possible solutions while maintaining efficiency for smaller problems. Backtracking operates dynamically during query evaluation and the order of clause evaluation is determined by the sequence in which clauses appear in the program, reflecting the depth-first nature of the process. This approach is both a strength and a limitation of Prolog, as it excels in solving symbolic computation and logical reasoning problems but may struggle with cases requiring breadth-first exploration [Körner et al., 2022].

Another characteristic of Prolog is its flexibility in the instantiation of arguments. Unlike other languages and paradigms, such as Haskell and functional programming, where the input-output relationship is strictly defined, Prolog allows arguments to be fully instantiated, partially instantiated, or even left uninstantiated. This capability supports a variety of query forms and makes Prolog highly versatile for exploring logical relationships and patterns.

The queries of the Listing 2.6, `ancestor(alice, Who)`, initiate the search for all descendants of `alice`. Prolog first matches the `ancestor/2` rule with a direct parent-child relationship and finds `Who = bob`. If the user requests more solutions, backtracking occurs, and Prolog explores the recursive clause, finding `Who = charlie` and then `Who = diana`. If no more solutions exist, the search terminates with a failure state, and Prolog reports `no`.

```
parent(alice, bob).
parent(bob, charlie).
parent(charlie, diana).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

?- ancestor(alice, diana).
true.

?- ancestor(alice, Who).
Who = bob ? ;
Who = charlie ? ;
Who = diana ? ;
no.

?- ancestor(Who, diana).
Who = charlie ? ;
Who = bob ? ;
Who = alice ? ;
no.

?- ancestor(Old, Young).
Old = alice, Young = bob ? ;
Old = alice, Young = charlie ? ;
Old = alice, Young = diana ? ;
Old = bob, Young = charlie ? ;
Old = bob, Young = diana ? ;
Old = charlie, Young = diana ? ;
no.
```

Listing 2.6: Backtracking using Flexible Argument Instantiation in Prolog

Prolog differs from paradigms like functional programming, which usually yield only one result, in that it can provide several responses through backtracking, as these queries illustrate. Users can more dynamically and exploratively examine the relationships embedded in the program by instantiating various combinations of arguments. This feature is essential to Prolog's declarative character and fits in with its logic-based framework. Its capacity to manage various solutions naturally adapts to situations that call for a thorough search or investigation of all options, which

increases its effectiveness in symbolic reasoning and problem-solving exercises.

Despite its advantages, Prolog has some drawbacks that prevent it from being widely used. First, programmers used to imperative or object-oriented techniques must adopt a new mentality due to its declarative and logic paradigm, which creates a steep learning curve. Furthermore, Prolog's scalability and performance are sometimes perceived as limited for applications involving vast amounts of data or computationally demanding activities [Körner et al., 2022, Warren et al., 2023].

SWI-Prolog[3] and SICStus[4] Prolog are two of the most popular Prolog interpreters, offering strong platforms for creating and running Prolog applications. First released in 1987, SWI-Prolog is especially well-known for being open-source and having a large library, which makes it a flexible option for both practical and scholarly applications [Wielemaker et al., 2010]. SWISH[5], an online interpreter that uses SWI-Prolog as its base and allows users to write and run Prolog programs directly in their browsers, is a notable addition to the SWI-Prolog ecosystem. On the other hand, SICStus Prolog is a popular option for large-scale systems and commercial applications due to its outstanding efficiency and industrial-grade dependability. It follows international standards set by the International Organization for Standardization (ISO), ensuring high reliability and making it ideal for managing large amounts of data and intricate applications [Carlsson and Mildner, 2010].

In addition to SWI-Prolog and SICStus Prolog, Yet Another Prolog[6] (YAP) is a Prolog system developed at the Department of Computer Science of the University of Porto. YAP is recognized for its high performance, achieved through optimizations of the Warren Abstract Machine (WAM) [Warren, 1983] and a dynamic indexing mechanism known as Just-In-Time Indexing (JITI), which enhances the efficiency of predicate lookups [Lempel et al., 2007, Costa et al., 2011]. Furthermore, YAP integrates both or-parallelism and tabling within a single logic programming environment, facilitating advanced execution strategies and efficient handling of recursive queries.

2.3 Summary

This chapter laid the groundwork for understanding Property-Based Testing by summarizing key ideas in programming paradigms and software testing. It began by going over the main categories of software testing, emphasizing the objectives, advantages, and disadvantages of various strategies. The chapter then looked at the basic operational features of programming languages. A variety of programming paradigms were also examined to demonstrate how they affected state management, code organization, and approaches to problem-solving. There are clear patterns, for example, the dominance of strong typing and the increasing support for multiple paradigms. It also underscores how Prolog stands apart with its logic paradigm and flexible execution model.

Table 2.1 provides an overview of the results concluded. The twelve languages selected for analysis reflect the most widely used programming languages, ordered chronologically by their

[3] Available online at: https://www.swi-prolog.org (last access on June, 2025)
[4] Available online at: https://SICStus.sics.se/index.html (last access on June, 2025)
[5] Available online at: https://swish.swi-prolog.org (last access on June, 2025)
[6] Available online at: https://www.dcc.fc.up.pt/~vsc/yap/ (last access on June, 2025)

year of creation, and chosen for their popularity and relevance[7], with additional consideration given to the legacy of languages like Lisp and Erlang, and the focus on Prolog as a core topic in this dissertation.

Table 2.1: Summary of Programming Languages Characteristics

Language	**Paradigm**	**Strong/ Weak Typed**	**Static/ Dynamic Typing**	**Comp./ Interp.**	**Polymorphism**
Lisp	Functional	Strong	Dynamic	Comp.	Yes
Prolog	Logical	Weak	Dynamic	Comp./ Interp.	Limited
C	Procedural	Strong	Static	Comp.	Limited
C++	Multi-paradigm	Strong	Static	Comp.	Yes
Erlang	Functional	Weak	Dynamic	Comp.	Limited
Haskell	Functional	Strong	Static	Comp.	Yes
Python	Multi-paradigm	Weak	Dynamic	Interp.	Yes
Java	OOP	Strong	Static	Comp./ Interp.	Yes
Ruby	Multi-paradigm	Strong	Dynamic	Interp.	Yes
JavaScript	Multi-paradigm	Weak	Dynamic	Interp.	Yes
C#	OOP	Strong	Static	Comp./ Interp.	Yes
Scala	Multi-paradigm	Strong	Static	Comp.	Yes

[7] Available online at: https://spectrum.ieee.org/top-programming-languages-2024 (last access on June, 2025)

Chapter 3

Property-Based Testing

This chapter presents the state-of-the-art Property-Based Testing tools and libraries, as well as the algorithms behind PBT, specifically for input generation and case shrinking. In addition, it explores the use and adoption of this technique in academic environments.

3.1 Tools and Libraries

The implementation of Property-Based Testing in a variety of programming paradigms demonstrates how adaptable it is to diverse language ecosystems. Haskell and Erlang are two functional languages that are notable for their strong PBT support. The first PBT library, QuickCheck[1] from Haskell, established fundamental ideas like automated test case generation and property definitions. A more recent development in the Haskell programming language is the introduction of the Hedgehog[2] and Falsify[3] libraries, which offer an alternative approach to property-based testing using integrated and internal shrinking during the input generation phase, respectively, and which will be further explored in Section 3.3. An open-source substitute called PropEr[4] in Erlang enhances PBT's capabilities by enabling parallel and stateful testing, which makes it especially appropriate for distributed systems. Additionally, the proprietary library QuviQ[5] QuickCheck offers similar features. These libraries provide dependable testing frameworks by utilizing the robust typing and logical paradigms of their respective languages. In contrast, while the Check-it[6] library offered similar functionality for Lisp, it is no longer maintained, and currently, there are no widely used or recent property-based testing libraries for the Lisp programming language.

[1] Available online at: https://hackage.haskell.org/package/QuickCheck (last access on June, 2025)
[2] Available online at: https://hackage.haskell.org/package/hedgehog (last access on June, 2025)
[3] Available online at: https://hackage.haskell.org/package/falsify (last access on June, 2025)
[4] Available online at: https://github.com/proper-testing/proper (last access on June, 2025)
[5] Available online at: https://www.quviq.com/documentation/eqc/overview-summary.html (last access on June, 2025)
[6] Available online at: https://github.com/DalekBaldwin/check-it (last access on June, 2025)

PBT frameworks in object-oriented and multi-paradigm languages offer flexibility to test a variety of application domains. Tools like jqwik[7] in Java, Hypothesis[8] and Schemathesis[9] in Python prioritize accessibility and interoperability with pre-existing testing routines, as demonstrated in Listing 3.1. The example below shows how the Hypothesis framework in Python integrates Property-Based Testing into an existing testing workflow, using the `@given` decorator to turn conventional test functions into property-based tests.

```
from hypothesis import given, strategies as st

@given(st.text())
def test_reverse_reverses_twice(string):
    assert reverse(reverse(string)) == string

@given(st.text())
def test_reverse_length(string):
    assert len(reverse(string)) == len(string)
```

Listing 3.1: PBT using the Hypothesis framework

The code presents two properties: `test_reverse_reverses_twice`, which verifies that double reversal of a string restores the original value, and `test_reverse_length`, which ensures that reversing a string preserves its length. In this example, the `@given` decorator facilitates accessibility by transforming conventional test functions into property-based tests without changing the structure of the existing test code. Additionally, using Hypothesis allows for interoperability with other popular testing frameworks as it can be integrated seamlessly without restructuring the existing test code. This makes it easier to adopt Property-Based Testing in projects that already use these frameworks, allowing PBT to be a natural extension to existing test workflows.

Libraries like jsverify[10] and Fast-Check[11] extend the reach of automated testing by bringing PBT concepts to dynamic, interpreted contexts, even though JavaScript has weaker typing. In the meantime, PBT is seamlessly integrated into contemporary software development pipelines by languages like C#, which make use of libraries like CsCheck[12]. ScalaCheck[13] for Scala adapts PBT concepts to their rich type systems.

Important tools for property-based validation are provided by implementations like RapidCheck[14], rantly[15] and theft[16], even in languages like C++, Ruby and C that are not typically

[7] Available online at: https://jqwik.net (last access on June, 2025)
[8] Available online at: https://hypothesis.readthedocs.io/en (last access on June, 2025)
[9] Available online at: https://schemathesis.readthedocs.io/en/stable/index.html (last access on June, 2025)
[10] Available online at: https://jsverify.github.io/#jsverify (last access on June, 2025)
[11] Available online at: https://github.com/dubzzz/fast-check (last access on June, 2025)
[12] Available online at: https://github.com/AnthonyLloyd/CsCheck (last access on June, 2025)
[13] Available online at: https://scalacheck.org (last access on June, 2025)
[14] Available online at: https://github.com/emil-e/rapidcheck (last access on June, 2025)
[15] Available online at: https://github.com/rantly-rb/rantly (last access on June, 2025)
[16] Available online at: https://github.com/silentbicycle/theft (last access on June, 2025)

linked with PBT. PrologCheck [Amaral et al., 2014] is a library that uses the logical programming language Prolog with YAP interpreter and emphasizes the applicability of the methodology in declarative situations. More recently, a randomized testing framework[17] for Prolog was developed, leveraging some characteristics of QuickCheck and implemented using SWI-Prolog.

Many PBT libraries are open source and constantly changing, giving developers the chance to modify them to suit particular requirements and systems, even though not all of them fully support sophisticated features like parallel or stateful testing[18]. This broad use highlights PBT's capacity to revolutionize a variety of programming languages

3.2 Input Generation Algorithms

Input generation is the first step in property-based testing, aiming to produce test cases tailored to the method under verification, with or without specific information about the data types of its arguments. To gather relevant documents and establish the state-of-the-art in existing algorithms, a search was conducted in three key research databases: the Association for Computing Machinery Digital Library (ACM DL)[19], IEEE Xplore[20] from the Institute of Electrical and Electronics Engineers (IEEE), and Scopus[21]. These databases were chosen because they index the primary conferences and journals in this field, ensuring comprehensive research coverage.

An article was considered relevant if it presented strategies, algorithms, or implementations adapted to specific programming languages, referencing the random or targeted generation or creation of inputs, cases, or examples for methods. The query focused on article abstracts because the abstract typically condenses the core content and techniques presented in the paper or article, providing a clear snapshot of the research focus. Additionally, requiring that property-based testing be referenced in the main body, and not solely in the abstract, ensured that the identified algorithms, even if not explicitly designed for PBT, could potentially support this testing strategy. This approach allowed for a broader search of algorithm-related articles, though with varying applicability to PBT. The quantitative results of the search conducted in October 2024, across the specified databases, are presented in Table 3.1.

The search resulted in the extraction of 38 relevant papers, of which 25 were unique. The analysis of the content revealed two major groups of strategies: random input generation, where input is generated solely based on the arguments of the method to be tested [Padhye et al., 2019, Mista and Russo, 2021, Goldstein et al., 2023b], and targeted input generation, where generated inputs follow specific heuristics and probabilities [Löscher and Sagonas, 2017, Lampropoulos et al., 2019]. Additionally, it was important to group articles based on foundations or implementations

[17] Available online at: https://github.com/nicoabie/quickcheck (last access on June, 2025)
[18] "The sad state of property-based testing libraries", available online at https://stevana.github.io/the_sad_state_of_property-based_testing_libraries.html (last accessed on June, 2025)
[19] Available online at: https://dl.acm.org (last access on June, 2025)
[20] Available online at: https://ieeexplore.ieee.org (last access on June, 2025)
[21] Available online at: https://www.scopus.com (last access on June, 2025)

Table 3.1: Quantitative results from search on Input Generation Algorithms

Research database	**Search command**	**No. results**	**No. relevant results**
ACM Digital Library	`Abstract:(random* OR target*) AND Abstract:(generat* OR creat*) AND Abstract:(input* OR case* OR exampl*) AND AllField:("property-based testing" OR "property based testing")`	51	10
IEEE Xplore	`("Abstract":"random*" OR "Abstract":"target*") AND ("Abstract":"generat*" OR "Abstract":"create*") AND ("Abstract":"input*" OR "Abstract":"case*" OR "Abstract":"exampl*") AND ( ("Full Text & Metadata":"property-based testing") OR ("Full Text & Metadata":"property based testing") )`	28	6
Scopus	`( ABS ( random* OR target* ) AND ABS ( generat* OR creat* ) AND ABS ( input* OR case* OR exampl* ) AND ALL ( "property-based testing" OR "property based testing" ) )`	115	22

in typed versus untyped languages, as one of the dissertation's objectives is to understand the challenges these algorithms face when handling type inference. The categorization of relevant papers found through research queries is presented in Table 3.2.

Table 3.2: Categorization of literature on input generation algorithms

	Typed language	Untyped language
Random generation	14	4
Targeted generation	5	2

It was observed that there is a gap in the literature on targeted generation technique [Löscher and Sagonas, 2017, Löscher and Sagonas, 2018]. Among these, only two papers reference solutions for systems developed in programming languages without static types [Lampropoulos et al., 2017, Lampropoulos et al., 2019]. This categorization of papers helped to guide and structure the state-of-the-art on algorithms used for input generation.

3.2.1 Random Input Generation

Claessen and Hughes [Claessen and Hughes, 2000] assert that random input generation is a strong substitute for systematic approaches in Property-Based Testing since it strikes a balance between usefulness and efficacy. Appropriate criteria, including making sure all control-flow paths are checked, are the foundation of systematic testing. But because of its higher-order functions, Haskell and other languages make it difficult to apply such requirements. Significant infrastructure would be needed to automate systematic procedures, such as compiler tweaks and constraint-solving strategies to produce particular test inputs or tasks that are challenging to perform and computationally costly for Haskell's symbolic datatypes. Furthermore, random testing provides fault detection rates comparable to partition testing, with any minor variations readily addressed by increasing the number of random test cases, according to empirical research by Duran and Ntafos [Duran and Ntafos, 1984], which is supported by Hamlet and Taylor [Hamlet and Taylor, 1990]. Strong coverage is ensured while preserving ease of use and accessibility by the fine-grained use of random testing in programs such as QuickCheck, where individual functions are checked separately.

In the field of random testing, knowing the difference between pseudo-random and truly random input creation is essential to assessing the efficacy and dependability of the testing procedure. James [James, 1990] defined really random numbers as originating from physical processes outside of computing systems. Deterministic algorithms, on the other hand, create pseudo-random numbers by simulating unpredictability while retaining an underlying predictability. The practicality and adaptability of these Pseudo-Random Number Generators (PRNG) make them popular in random testing. Due to this fact, the generated inputs are not actually random but rather pseudo-random. Function 3.1 presents the formal definition of a PRNG.

$$g : S \rightarrow \tau \tag{3.1}$$

A PRNG is a function g that transforms a seed from the set S into an output of type τ, where τ is the data type of the outputs produced and S is the set of seeds used to initialize the generator.

According to this definition, PRNG are deterministic, meaning that the initial seed determines the entire output sequence. As a result, PRNG make controlled and repeatable testing scenarios possible. The same seed can be used to generate identical input data sequences [de Vries, 2023], guaranteeing consistency between test runs and making debugging and result verification easier [Bhattacharjee et al., 2018].

Programming languages usually include PRNG in their standard libraries for real-world applications. To add variation and prevent predictable results, these libraries pull seeds from a variety of sources, typical sources include operating system-provided randomness (from `/dev/random` or `/dev/urandom` on Unix-like systems), system time (for example, the number of milliseconds since a set epoch), or even hardware-based randomization where supported [Padhye et al., 2019]. Although users can explicitly choose a specific seed to facilitate reproducibility in testing settings, the libraries leverage these sources to guarantee that the initial seed for a PRNG is appropriately random for the majority of applications. As a result, it is reasonable to infer that any algorithm employed in this subsection to generate random inputs will really produce pseudo-random values. For simplicity's sake, they will still be called randoms.

In the context of Property-Based Testing, a `generator` is a mechanism that produces random inputs to test the properties defined in a given program [Runciman et al., 2008, Aichernig et al., 2017, De Angelis et al., 2019, Ziat et al., 2021]. These generators are a critical component as they define the structure and distribution of those inputs. Claessen and Hughes, in their implementation of QuickCheck [Claessen and Hughes, 2000], created an interface using Haskell's overloading mechanism, known as type classes, to define the polymorphic `Arbitrary` class for random generators. This polymorphism allows the same interface to support random generation for multiple types, each with its own implementation.

```
1 newtype Gen a = Gen (Rand -> a)
2 choose :: (Int, Int) -> Gen Int
3
4 class Arbitrary a where
5     arbitrary :: Gen a
6
7 instance Arbitrary Int where
8     arbitrary = choose (-20, 20)
```

Listing 3.2: Defining polymorphic generators using the Arbitrary type class in Haskell

Listing 3.2 defines a generator for integers, `Int` type, between -20 and 20, where `Rand` is a random number seed. The `choose` function is used to specify the range, showcasing how the `Arbitrary` type class supports polymorphic random generation.

Simple, predefined generators, like `String` and `Int`, and custom, user-defined generators are two types of possible generators. For popular base types, simple generators are easily accessible and frequently adequate for simple instances. Custom generators, where developers specify how values for a certain type should be created, are necessary for more complicated circumstances [Lampropoulos et al., 2019, Goldstein et al., 2023a].

`Combinators`, also known as User-Defined Types, are frequently provided by PBT frameworks to make the process of creating such intricate generators easier [Paraskevopoulou et al., 2015, Mista and Russo, 2021, de Vries, 2023]. Combinators are higher-order structures that capture the logic of random data creation by allowing the synthesis of smaller generators into more

complex ones. In addition to minimizing redundancy, this abstraction guarantees generator definitions are clear and reusable. Narrowing, an idea taken from functional logic programming and introduced by Antoy et al. [Antoy et al., 2000], refers to the lazy construction of a random data structure while navigating the definition of the predicate that the structure must satisfy, making it particularly effective in PBT techniques. It concentrates solely on the essential components of the structure that are pertinent to the predicate, eliminating needless computation and guaranteeing that the generated data satisfies the appropriate requirements. This method enables the more efficient construction of complicated data structures.

Combining this abstraction with Abstract Data Types (ADT), which enable the construction of intricate, organized data with clearly defined operations, makes it even more potent [Chen et al., 2017]. Lampropoulos et al. [Lampropoulos et al., 2018] address recursive logic using generators from a Property-Based Testing library for the Coq[22] programming language called QuickChick[23], which is based on QuickCheck. Since ADT offer a formal method for defining and working with recursive data structures [Gissurarson et al., 2022, Graeff et al., 2024], such as binary trees, while preserving invariants, they use it to generate a first approach for a random binary tree as demonstrated in Listing 3.3.

```
1 Fixpoint genTree : G Tree :=
2     oneOf [ ret Leaf ;
3             do! x <- arbitrary ;
4             do! l <- genTree ;
5             do! r <- genTree ;
6             ret (Node x l r) ].
```

Listing 3.3: Generating a random binary tree using QuickChick's combinators and Narrowing

Recursively defining a generator for a binary tree, the `genTree` function combines recursive nodes, `Node x l r`, with the base case, `Leaf`. Narrowing guarantees that the tree is created slowly in this situation, beginning at the root and growing only as necessary. It enables the generator to concentrate on the essential elements, `x, l` and `r` when building a node. This ensures efficient and legitimate tree creation by producing only the necessary subtrees recursively without precalculating extraneous portions of the tree.

Despite their benefits, combinators have drawbacks. Their intricacy is a major obstacle: when building complicated generators, wrong implementation or carelessness may result in the introduction of flaws, rendering the generation logic inaccurate or unreliable [Lampropoulos et al., 2017, Mista and Russo, 2021]. Ensuring termination, making sure the generation process finally ends and does not produce non-terminating structures or indefinite recursion, might be quite troublesome. Listing 3.4 supplied by Lampropoulos et al. [Lampropoulos et al., 2018] serves as an example of this strategy.

[22] Available online at: https://coq.inria.fr/about-coq (last access on June, 2025)
[23] Available online at: https://github.com/QuickChick/QuickChick (last access on June, 2025)

```
Fixpoint genTreeSized (size : nat) :=
    match size with
        | O => ret Leaf
        | S size' => freq [ (1, ret Leaf)
            ; (size, do! x <- arbitrary;
                     do! l <- genTreeSized size';
                     do! r <- genTreeSized size';
                     ret (Node x l r)) ]
    end.
```

Listing 3.4: Ensuring termination in binary tree generation with size limits

In Property-Based Testing frameworks, combinators are by definition frequently built on generators in which every conceivable value or configuration has an equal chance of being chosen. A wide exploration of the input space is ensured by this uniform distribution, which maximizes the diversity of the test cases produced. Evenly creating every potential constructor of ADTs ensures that every scenario has an equal chance of being evaluated, offering a thorough understanding of the system's behavior. Uniform distribution, however, might not always be the best course of action. Even though they are less common, some uncommon configurations could have important characteristics or edge cases that need additional consideration during testing. For example, a strictly uniform distribution can miss testing specific configurations that are unlikely yet essential to the program's correctness [Mista and Russo, 2021].

Amaral et al. [Amaral et al., 2014] in PrologCheck present the idea of a frequency parameter, which enables programmers to provide probabilities for the generator selection process.

```
Gen1 :- listOf(int).
Gen2 :- listOf(float).

frequency (FGL, A, S):-
    checkFreqWeights (FGL, FIL, Cap), choose (1, Cap, I, S),
    nth (I, FIL, GenA), call (GenA, A, S).

frequency([{4, Gen1}, {1, Gen2}], A , S)...
```

Listing 3.5: Using the frequency parameter in PrologCheck to bias generator selection

Listing 3.5 provides an example of how this was implemented and how the frequency parameter biases the generator selection process according to predetermined weights. This flexibility allows testers to fine-tune the distribution of test cases, favoring scenarios with the highest relevance to the attributes being tested. Each generator, `Gen1` and `Gen2`, is given a weight in the `frequency/3` predicate; the higher the weight, the more likely the generator is to be selected. The `choose/4` predicate favors generators with larger weights by choosing a generator index depending on the cumulative weight.

As noted in the case of PrologCheck, the absence of types among variables does not prevent libraries from adopting this strategy. In fact, to test a component, it is always necessary to specify the type of data expected in the arguments. In Python, a dynamically typed language, the situation is very similar to the Hypothesis framework, always allowing the adoption and creation of new data structures [Goldstein et al., 2023a, de Vries, 2023].

```
>>> import hypothesis.strategies as st

>>> x = st.deferred(lambda: st.booleans() | st.tuples(x, x))

>>> x.example()
(((False, (True, True)), (False, True)), (True, True))

>>> x.example()
True
```

Listing 3.6: Recursive Data Generation with Hypothesis

The code of Listing 3.6 demonstrates how to create recursive data structures using Hypothesis techniques. The method alternates between producing tuples with more instances of the same structure and booleans, displaying both straightforward and intricately nested outputs.

All the PBT frameworks presented in the Table 2.1, regardless of whether the adopted language is statically or dynamically typed, support random input generation.

3.2.2 Targeted Input Generation

Löscher and Sagonas [Löscher and Sagonas, 2017] introduce Targeted Property-Based Testing (TPBT) as a significant enhancement to traditional Property-Based Testing. The limits of random PBT are first demonstrated by addressing a real-world situation involving network graphs. In the example, a network graph with a set of nodes is used to demonstrate the property stating that the shortest path between a designated node and any other node in the graph must not exceed half the total number of nodes. However, the difficulty of finding counterexamples to this condition, which entails creating networks with extremely specialized topologies, cannot be adequately addressed by random testing since it depends just on chance.

Targeted input generation is the process of guiding input creation using a search strategy to optimize for properties likely to falsify the test. When counterexamples are rare or unlikely to be generated by a generic data combinator, this method is crucial for effectively exploring large input spaces. TPBT depends on four essential elements:

- **Targeted Generator** (TG): allow the creation of inputs tuned to specific testing needs;
- **Utility Value** (UV): numerical indications that direct the search toward inputs with a larger chance of failure by showing how close an input is to falsifying a property;

- **Neighboring Function** (NF): convert an input into a close variant while maintaining its structural properties and invariants. The likelihood of finding counterexamples is further increased by these functions, which allow inputs to be fine-tuned around interesting candidates found during the search;
- **Search Strategy** (SS): controls how the input space is explored in order to maximize or minimize utility values.

To maximize the discovery of counterexamples, the general algorithm described in Algorithm 1 incorporates these components into an iterative procedure.

Algorithm 1 General Targeted Input Generation Algorithm

Require: Property P, Initial Input I_0, Targeted Generator TG, Search Strategy SS, Utility Function UV
Ensure: Counterexample I falsifying P or confirmation of robustness
  $I \leftarrow I_0$
  while maximum number of inputs not reached **and** $P(I) \neq$ `false` **do**
    $I \leftarrow SS(I, TG, NF, UV)$
  end while
  if $P(I) =$ `false` **then**
    return I
  else
    return "Property holds for tested inputs"
  end if

The algorithm starts with an initial input, which can be generated by random input creation algorithms, and iteratively improves it by creating new candidates. However, one of the major challenges of this type of algorithm is finding an appropriate SS. Löscher and Sagona [Löscher and Sagonas, 2017] implemented and tested TPBT according to two algorithms: Hill Climbing and Simulated Annealing.

Hill Climbing (HC) creates a neighboring input and calculates its utility value from an initial input. The new input takes the place of the existing one if its UV is higher. In order to prevent the search from digressing needlessly, neighboring functions direct it by concentrating on areas of the input space that are more likely to produce counterexamples. When a convex problem has a clear path to an optimum, this method works especially well. The Algorithm 2 shows the general Hill Climbing Search Strategy.

Despite its simplicity, HC is less appropriate for complex optimization scenarios since it frequently fails to break out of local optima [Chalup and Maire, 1999]. There are no mechanisms in the algorithm to investigate worse options that could result in a global optimum. It might therefore converge too soon to less-than-ideal solutions, particularly in non-convex or extremely irregular search spaces.

Simulated Annealing (SA) extends Hill Climbing by incorporating a mechanism to escape local optima [Chalup and Maire, 1999]. SA, which draws inspiration from the physical annealing process, accepts worse solutions with a decreasing probability as the temperature parameter cools

Algorithm 2 Hill Climbing Search Strategy

Require: Initial Input I, Targeted Generator TG, Utility Function UV, Neighboring Function NF
Ensure: Next Input I with optimized utility

```
currentInput ← I
currentUtility ← UV(currentInput)
while stopping condition not met do
    newInput ← NF(currentInput, TG)
    newUtility ← UV(newInput)
    if newUtility > currentUtility then
        currentInput ← newInput
        currentUtility ← newUtility
    end if
end while
return currentInput
```

over time. This raises the algorithm's chances of discovering a global optimum by enabling it to examine a larger portion of the input space. Iteratively, SA generates neighboring inputs based on a probability function connected to the current temperature after starting with a random input and evaluating its Utility Value. The search gradually lowers the temperature and becomes more focused over time, striking a balance between optimization and exploration. The Algorithm 3 shows the general Simulated Annealing Search Strategy.

Algorithm 3 Simulated Annealing Search Strategy

Require: Initial Input I, Targeted Generator TG, Utility Function UV, Neighboring Function NF, Initial Temperature T, Cooling Schedule C
Ensure: Next Input I with optimized utility

```
currentInput ← I
currentUtility ← UV(currentInput)
while stopping condition not met do
    newInput ← NF(currentInput, TG)
    newUtility ← UV(newInput)
    if newUtility > currentUtility or e^((newUtility − currentUtility)/T) > random(0,1) then
        currentInput ← newInput
        currentUtility ← newUtility
    end if
    T ← C(T)
end while
return currentInput
```

TPBT significantly reduces the number of tests needed to find counterexamples by using Utility Values and Search Strategies to improve the likelihood of finding bugs in large input spaces. However, it also necessitates that testers apply search strategies and comprehend the full context. Custom neighborhood functions also take more work to implement. Due to their reliance on the property itself, the context, and the data input structure, these neighborhood functions make this class of input generation strategies less flexible and scenario-adaptive. In a more recent study,

Löscher and Sagona [Löscher and Sagonas, 2018] present an automated method for TPBT in order to get around the difficulty of creating custom NF. Although the framework only requires testers to specify a data generator, it may not be appropriate for all kinds of properties or complex input structures, and it may result in less precise control over the input generation process.

Currently, there are not many property-based testing frameworks that use targeted input generation. The research provided by Löscher and Sagona [Löscher and Sagonas, 2017, Löscher and Sagonas, 2018] resulted in the tool Target, which is now fully integrated into the TPBT framework PropEr[24] in Erlang. Schemathesis[25], for the Python programming language, implicitly uses the Hypothesis framework and provides an Application Programming Interface (API) to guide data generation towards certain pre-defined goals, which can be customized.

Listing 3.7 highlights the creation of a targeted generator that promotes the generation of inputs where the response progressively increases in size. Given the implementations in dynamically typed languages like Erlang and Python, the absence of static typing does not present a fundamental obstacle to the implementation of targeted input generation.

```
1 import schemathesis
2
3 @schemathesis.target
4 def new_target(context: any) -> float:
5     return float(len(context.response.content))
```

Listing 3.7: Example of using a targeted generator in Schemathesis to bias input generation

According to a study on Python projects by Corgozinho et al. [Corgozinho et al., 2023], more sophisticated methods such as targeted generators were underutilized, whereas simpler types of input generation were preferred. Creating generators that meet property preconditions requires a lot of work and complexity, which developers find tiresome and disruptive to their workflow. This slow uptake of cutting-edge methods demonstrates the disconnect between TPBT potential and real-world implementation. Because of the learning curve or a lack of understanding, developers rarely take full advantage of frameworks [Goldstein et al., 2024], which provide customizable choices for targeted generation. Focused input creation is still a neglected field, even with the benefits of identifying edge cases or scaling scenarios.

Table 3.3 summarizes the main categories of input generation algorithms discussed above, dividing them into random and targeted approaches. For each category, key advantages and disadvantages are highlighted to provide a clear overview of their respective strengths and limitations.

[24] Available online at: https://github.com/proper-testing/proper (last access on June, 2025)

[25] Available online at: https://schemathesis.readthedocs.io/en/stable/index.html (last access on June, 2025)

Table 3.3: Overview of Input Generation Algorithms

Type	Advantages	Disadvantages
Random	Simple to implement. Broad exploration of input space. Fast generation of diverse cases.	May produce many irrelevant or invalid inputs. Low guarantee of targeted coverage. Difficult to focus on critical code sections.
Targeted	Directs generation toward specific interesting cases. Improves efficiency in discovering complex bugs. Can preserve invariants during generation.	More complex to implement. Requires prior knowledge of properties or invariants. May leave parts of input space unexplored.

3.3 Shrinking Algorithms

Shrinking is the second step in property-based testing, involving the systematic reduction of the corresponding input, simplifying it to its most basic form and identifying minimal, reproducible test cases [Lo et al., 2020]. To gather relevant documents for obtaining the state-of-the-art algorithms, a search was conducted based on the three previously mentioned research databases.

An article was considered relevant if it presented algorithms, whether adapted or not for specific programming languages, referencing shrinking, reduction, or simplification, as well as Property-Based Testing. The query, as in the previous subsection, was focused on paper abstracts, which summarize the techniques employed throughout the article and required a reference to Property-Based Testing in the main body. This approach allowed for a broader search of shrinking-related papers, though with varying applicability to PBT. The quantitative results of the search conducted in October 2024, across the specified databases, are presented in Table 3.4.

The search yielded a total of 29 relevant papers, 18 of which were unique. As in the previous subsection, it was necessary to categorize the papers found in order to guide the foundations of shrinking algorithms and establish the state-of-the-art techniques. Through content analysis, it was observed that there are three main types of input shrinking: external shrinking, where this step is performed after the input generation phase as a separate process [Claessen and Hughes, 2000, Blanco et al., 2019, Lo et al., 2020, Krook et al., 2023], and integrated and internal shrinking, where case reduction occurs during creation in a single step [MacIver and Donaldson, 2020, de Vries, 2023]. There is also a distinction between algorithms and implementations regarding statically typed and dynamically typed programming languages. Based on these differences, the resulting papers were grouped, with the outcome presented in Table 3.5.

There is a clear lack of information in the literature regarding internal shrinking and algorithms implemented in untyped languages. This disparity highlights the need to invest in such solutions for the development of new tools using property-based testing, which is a primary objective of this dissertation.

Table 3.4: Quantitative results from search on Shrinking Algorithms

Research database	**Search command**	**No. results**	**No. relevant results**
ACM Digital Library	`Abstract:(shrink* OR reduc* OR simplif*) AND AllField:("property-based testing" OR "property based testing")`	53	10
IEEE Xplore	`("Abstract":"shrink*" OR "Abstract":"reduc*" OR "Abstract":"creat*") AND (("Full Text & Metadata":"property-based testing") OR ("Full Text & Metadata":"property based testing"))`	55	5
Scopus	`( ABS (shrink* OR reduc* OR simplif*) AND ALL ( "property-based testing" OR "property based testing" ))`	108	14

Table 3.5: Categorization of literature on shrinking algorithms

	Typed language	Untyped language
External shrinking	7	3
Integrated shrinking	2	1
Internal shrinking	3	2

3.3.1 External Shrinking

External shrinking keeps the processes of creating test cases and shrinking them apart. In this method, a generator only generates input values, while a separate shrinker tries to make any counterexamples discovered during testing simpler. This division results in a modular design that allows testers to optimize both steps in parallel, an approach that forms the foundation for the implementation of some PBT libraries such as QuickCheck[26] and ScalaCheck[27].

The shrinking process needs recursive or iterative algorithms to explore smaller versions of the test case. Claessen and Hughe [Claessen and Hughes, 2000] introduced this technique in QuickCheck's implementation, where a shrink function receives a value that falsifies a certain

[26] Available online at: https://hackage.haskell.org/package/QuickCheck (last access on June, 2025)
[27] Available online at: https://scalacheck.org (last access on June, 2025)

property and returns a list of smaller candidates. This is achieved using a shrink function associated with the data type being tested, and these candidates are tested against the property until the minimal counterexample is found. For instance, the shrinker may systematically reduce a random list by eliminating elements, reducing their sizes, or substituting simpler values for elements if the list fails a property. Listing 3.8 combines a shrinker `shrinkWord` that minimizes test cases with a generator `genWord` for unsigned integers. The shrinking logic, which produces candidates in the range `[0..n - 1]`, is compatible with Haskell's `Word` type, which represents non-negative integers. The combinator `forAllShrink` then connects the generator and shrinker, giving an example in QuickCheck that guarantees the sum of a list's elements is always greater than or equal to its maximum value.

```
genWord :: Gen Word
genWord = arbitrary

shrinkWord :: Word -> [Word]
shrinkWord 0 = []
shrinkWord n = [0 .. n - 1]

prop_SumGreaterThanElements :: [Word] -> Bool
prop_SumGreaterThanElements xs = sum xs >= maximum xs

main :: IO ()
main = quickCheck (forAllShrink genWord shrinkWord
    prop_SumGreaterThanElements)
```

Listing 3.8: Shrinking in QuickCheck for the sum property

For composite data types where each component has its own generator and shrinker, external shrinking is particularly helpful [Pike, 2014]. In Listing 3.9, the `shrinkPair` function combines the shrinking strategies for individual components to create a unified shrinker for pairs.

```
shrinkPair :: (a -> [a]) -> (b -> [b]) -> (a, b) -> [(a, b)]
shrinkPair shrinkA shrinkB (x, y) =
  [(x', y) | x' <- shrinkA x] ++ [(x, y') | y' <- shrinkB y]
```

Listing 3.9: Shrinking composed types in QuickCheck

There is a significant limitation in this technique when using compositional generators [de Vries, 2023]. For instance, consider a generator that only produces even unsigned integers. The generator `genEvenWord` can be composed by mapping over `genWord`, as shown in Listing 3.10.

```
genEvenWord :: Gen Word
genEvenWord = fmap (*2) genWord
```

Listing 3.10: Compositional generators

Although the base data type remains the same, using the previous `shrinkerWord` may produce invalid shrunk values within the context because the invariant is not maintained. For every mapping `f:a -> b`, there must exist a mapping `f':b -> a` before computing any shrinking process. It is not possible to adopt default shrinker functions, forcing testers to implement their own shrinker, often duplicating logic.

In a more recent study, Amaral et al. [Amaral et al., 2014] highlighted also the inherent difficulty of programming external shrinkers that focus on data types in dynamically typed languages like Prolog. In the PrologCheck implementation, shrinkers are calls to the matching generator. A generator is called with the value to shrink and a variable to hold the list of shrunk values in order to start shrinking. Removing an element is a valid example of a shrinking behavior for lists, as shown in Listing 3.11.

```
genL(GenA, A, Size) :- listOf(GenA, A, Size).
genL(GenA, L, shrink, Shrs) :- shrL(L, Shrs).

shrL([], []).
shrL([A], [[]]):- !.
shrL([A|AS], [AS|Shrs]) :-
    shrL(AS, Shrs1),
    maplist(cons(A), Shrs1, Shrs).

cons(X, XS, [X|XS]).
```

Listing 3.11: Shrinking implementation example in PrologCheck

To implement the shrink behavior for this particular type, the tester must wrap combinators since the majority of PrologCheck combinators lack a shrinking method. The shrinking of many combinators is set to an empty list of shrunk values by default, since it is difficult to determine what a good shrink is for values produced by a random choice amongst generators.

The external shrinking approach provides control and flexibility but requires careful attention from testers to maintain consistency between the generator and shrinker [Pike, 2014, Chen et al., 2022]. Since the shrinking is implemented as a separate function for each data type, the shrinker must preserve the invariants of the original generator, a requirement that can be challenging to enforce in more intricate scenarios; otherwise, it risks producing invalid or irrelevant inputs [de Vries, 2023]. The logic for generating values often overlaps with the logic for shrinking them, leading to potential redundancy.

3.3.2 Integrated Shrinking

Integrated shrinking combines the input generation and shrinking processes into a single abstraction. Instead of generating just a possible input and then shrinking it in a separate flow, the generator produces a tree structure, where the root is the generated value, and its children represent

successive, recursive shrink steps [de Vries, 2023]. This approach is used in libraries like QuviQ[28] QuickCheck for the Erlang programming language and Hedgehog[29] for Haskell.

This approach addresses the main weaknesses of external shrinking, since it avoids duplication and redundancy between generators and shrinkers of the same data type, as demonstrated in the Hedgehog implementation in Listing 3.12.

```
data Tree a = Node a [Tree a]
data Gen a = Gen (PRNG -> Tree a)

genWord :: Gen Word
genWord = Gen $ \seed ->
  let val = randomInt seed
  in Node val [Node i [] | i <- [0..val-1]]

genEvenWord :: Gen Word
genEvenWord = fmap (*2) genWord

genPairs :: Gen (Word, Word)
genPairs = Gen $ \seed ->
  let Node val1 shrinks1 = runGen genWord seed
      Node val2 shrinks2 = runGen genWord (seed + 1)
  in Node (val1, val2) [Node (i1, i2) [] | (i1, i2) <- zip shrinks1
     shrinks2]
```

Listing 3.12: Integrated Shrinking using Hedgehog

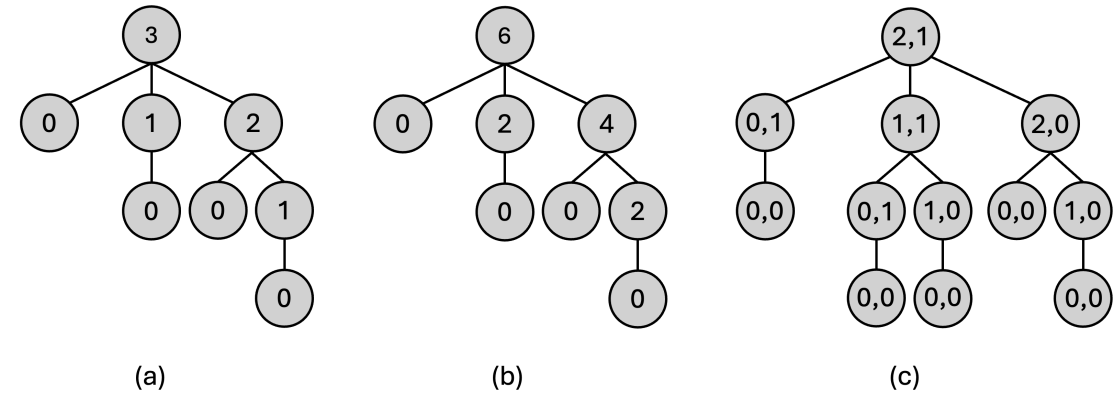

Figure 3.1: Integrated Shrinking Trees. Adapted from [de Vries, 2023].

[28] Available online at: https://www.quviq.com/documentation/eqc/overview-summary.html (last access on June, 2025)

[29] Available online at: https://hackage.haskell.org/package/hedgehog (last access on June, 2025)

The integration of shrinking into the generation process offers two additional major advantages[30]. First, shrinking composes seamlessly. One possible shrinking tree for `genEvenWord`, shown in Fig. 3.1 (b), can be viewed as an extension of the shrinking tree for `genWord`, presented in Fig. 3.1 (a). This ensures that the shrinking process satisfies the same invariants as the generation process. The approach supports compositional operations as well, such as combining generators to produce complex structures like pairs or lists. For example, `genPairs` can generate a shrinking tree as shown in Fig. 3.1 (c).

Integrated shrinking presents its biggest obstacle in monadic composition. This situation frequently occurs when one component of a generated structure depends on the outcome of another, and the generator's behavior relies on previously generated values, such as when creating a list with random characters and a random length. Listing 3.13 presents an example of monadic composition.

```
randomList :: Gen [Char]
randomList = do
    n <- word
    replicateM n char
```

Listing 3.13: Monadic composition in Hedgehog

In this case, the list length `n` is generated first, followed by the random characters. This dependency introduces the need to compose the generators, and the issue arises when attempting to apply shrinking to this composition. The shrinker may systematically reduce a random list by eliminating elements, reducing the list size, or substituting simpler values. Given the basic shrinking process demonstrated in Fig. 3.2 (a), the algorithm results in a recursive tree, as illustrated in Fig. 3.2 (b).

This process will be inefficient, as shrinking the first generator may require recomputing subsequent values, discarding progress and causing redundancy. Moreover, there can be no inherent relation between nodes in different subtrees, potentially leading to illogical shrinkage sequences.

According to Edsko de Vitres [de Vries, 2023], there are two potential approaches to handle this tree structure: left-biased join and right-biased join. Left-biased join shrinks the list length first, as shown in Fig. 3.2 (c), while right-biased join shrinks the list elements first, as presented in Fig. 3.2 (d). Both methods have the flaw of failing to revisit the list length after the list elements have been shortened, which breaks the invariant between them. Because of the improper maintenance of the link between the list length and its elements, the shrinking process loses coherence. In integrated shrinking, structural invariants are always broken and inefficiencies are introduced by the recomputation needed to preserve the relationship between dependent generators.

[30] Available online at: https://hypothesis.works/articles/integrated-shrinking/ (last access on June, 2025)

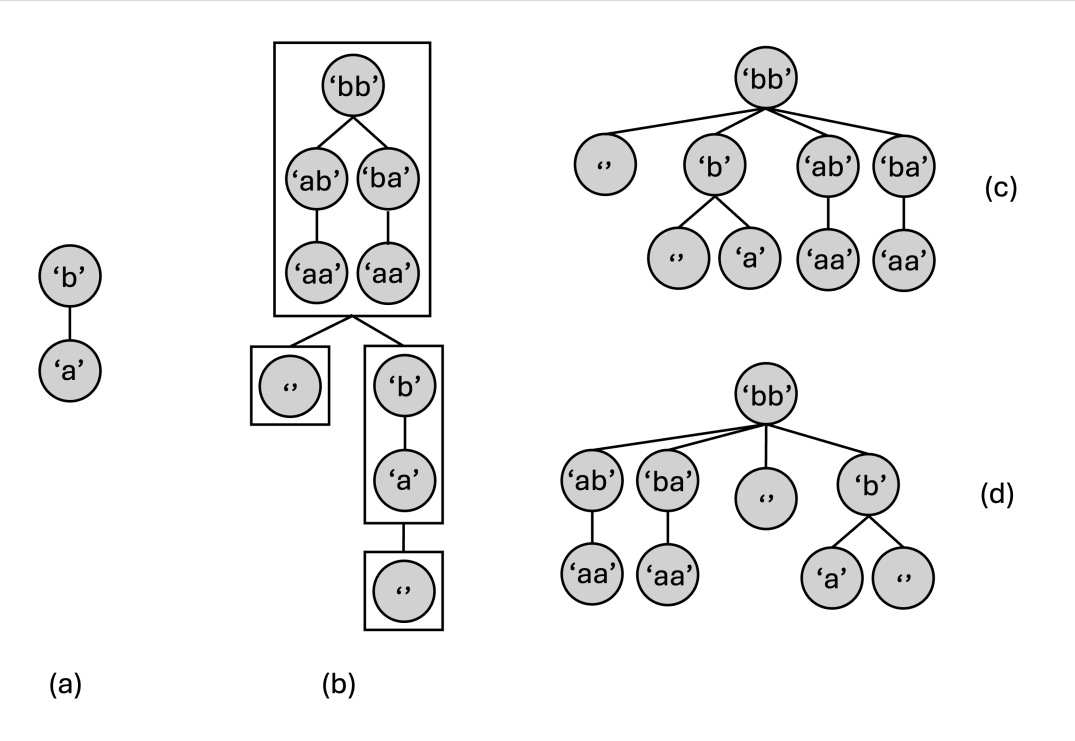

Figure 3.2: Monadic Composition in Shrinking Trees. Adapted from [de Vries, 2023].

3.3.3 Internal Shrinking

Internal shrinking is a method of test-case reduction used in the Hypothesis[31] library for Python since 2016 and in the Falsify[32] library for the Haskell programming language. It relies on the linear use of a splittable Pseudo-Random Number Generator (PRNG), a type of random generator that can be divided into independent sub-generators. Listing 3.14 presents a simple use of linear PRNG in Haskell.

```
genPair :: PRNG -> Int -> (Int, Int)
genPair prng seed = (int1, int2)
  where
    (prng1, prng2) = split prng
    int1 = prng1 seed
    int2 = prng2 seed
```

Listing 3.14: Linear use of PRNG in Haskell

In parallel or modular operations, where various program components need separate, non-overlapping random streams, each sub-generator generates a distinct series of random numbers [Goldstein and Pierce, 2022]. Because the splitting process is deterministic, repeatability is guaranteed as it consistently yields the same sub-generators given the same initial seed.

[31] Available online at: https://hypothesis.readthedocs.io/en/latest/ (last access on June, 2025)
[32] Available online at: https://hackage.haskell.org/package/falsify (last access on June, 2025)

Internal shrinking automatically generates smaller test cases by adjusting the random generator's seed [MacIver and Donaldson, 2020], eliminating the need for an external reducer, as test-case reduction is integrated into the generator. In other words, the shrinker is assumed to have access to the generator's initial random choices. The simplified test case will preserve any properties that the generator is specifically designed to guarantee.

Both the seed and the output of PRNG libraries in various programming languages have fixed bit widths. The seed itself can be viewed as an array of bits, also referred to as a `sample stream`, which drives the generation process. In a real-world scenario where the generator creates a list of random size and elements, some bits identify the elements, while others decide the length of the list. Because the underlying choice sequence size, also known as the sample stream, is the same for all potential values, this shrinking can only be accomplished by altering the contents of the bits rather than by drawing fewer bits. The length of the list or its elements will change if these bits are changed. Thus, in addition to preserving the advantages of integrated shrinking, this technique addresses the problem of monadic composition highlighted in Fig. 3.2, and the structural invariants are always maintained. The difference in data flows between this approach and the previous two can be visualized in Fig. 3.3.

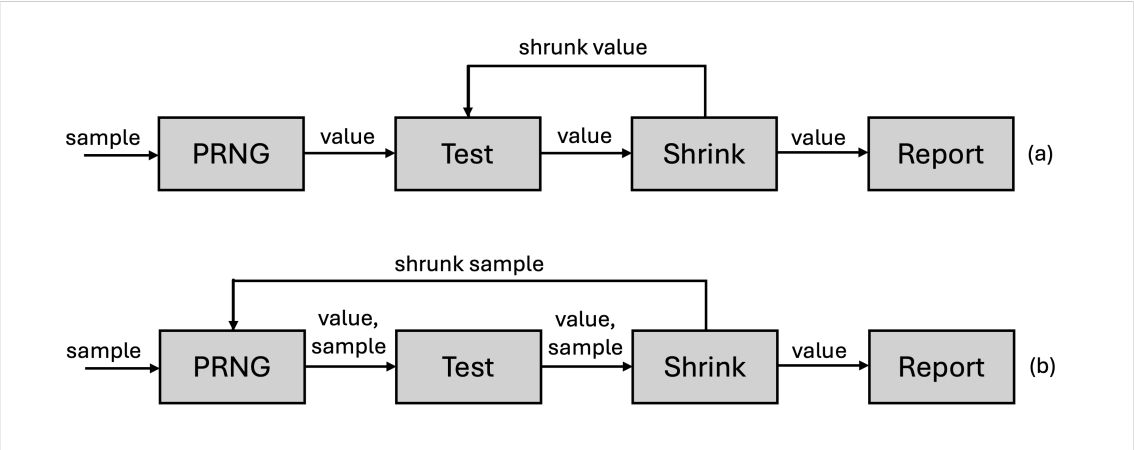

Figure 3.3: Overview of External and Integrated Shrinking (a) and Internal Shrinking (b) flows

This algorithm used the shortlex order[33] to shrink the sample stream. It is a sorting technique that uses lexicographic order within sequences of the same length and gives priority to shorter sequences. This guarantees a methodical decrease in inputs during sample shrinkage, hence reducing complexity in a predictable manner. For example, shortlex tries shorter sequences like `[1, 1]` or `[0, 1]` before lexicographically different sequences of the same length like `[0, 0, 1, 1]` when decreasing a failed bit sequence like `[1, 1, 0, 1]`. This method guarantees the finding of a few, understandable counterexamples while maintaining logical progression and avoiding chaotic sample redistribution.

The implementation of Falsify in a more recent work by Edsko de Vitres [de Vries, 2023] follows a methodology similar to Hypothesis. However, instead of manipulating a sample stream,

[33] Available online at: `https://en.wikipedia.org/wiki/Shortlex_order` (last access on June, 2025)

it manipulates sample trees. A sample tree is a data structure shown in Listing 3.15 with the simplest sample represented at the node and increasingly more sophisticated versions of this sample displayed at the branches. Shrinking then entails choosing a branch that is nearer the root.

```
data STree = STree Sample STree STree
           | Minimal

data Gen a = Gen { run :: STree -> (a, [STree]) }

genWord :: Gen Word
genWord = Gen $ \(STree sample l r) ->
  ( value sample, [ STree (Sample s') l r | s' <- shrinkWord (value sample
     ) ])
```

Listing 3.15: Sample Tree construction

A generator for numbers might go through a tree in which each node divides the range of potential values into smaller intervals. Lazy lists and functions are examples of limitless data structures that this structure can handle, because the tree can shrink each component separately and without interfering.

Hypothesis and Falsify assume the shrinker has access to the generator's original random choices, so it fails when shrinking is separated from generation or for values that the library did not generate. Moreover, according to MacIver and Donaldson [MacIver and Donaldson, 2020], the performance is not the main benefit of internal shrinking. Although it can be a little slower, this test case reduction yields results that are somewhat better than those of other property-based testing libraries.

Table 3.6 summarizes the main categories of shrinking algorithms discussed above, dividing them into external, integrated, and internal approaches. For each category, key advantages and disadvantages are highlighted to provide a concise overview of their strengths and limitations.

Table 3.6: Overview of Shrinking Algorithms

Type	Advantages	Disadvantages
External	Provides control and flexibility. Separate implementation allows independent optimization and clear separation of concerns.	Requires synchronization between generator and shrinker. Risks in producing invalid inputs if invariants are not preserved. Logic duplication.
Integrated	Avoids redundancy by combining generation and shrinking. Ensures shrinking respects generation invariants. Supports compositional shrinking for complex data structures.	Challenging monadic composition. Inefficiencies due to recomputation. Possible broken invariants in dependent data structures.

Type	Advantages	Disadvantages
Internal	Shrinking integrated into generator via seed manipulation. Maintains structural invariants even with complex dependencies. Deterministic and repeatable shrinking process.	Only works if shrinker has access to original generator choices. Less effective if shrinking is decoupled from generation. Potentially slower performance.

3.4 Academic Landscape

In academic fields, Property-Based Testing has been increasingly recognized as a useful strategy for raising the caliber of programming exercise evaluation and encouraging students to comprehend testing and specification principles more thoroughly.

PBT is used by an automated framework designed to assess programming exercises, leveraging this testing technique to validate both the functional correctness and algorithmic complexity of student submissions [Benac Earle et al., 2016]. It enables the automatic generation of a large number of test cases from a property model, which are then used to test the submitted programs. A composite grade is also determined by evaluating non-functional indicators like cyclomatic complexity. When the framework was used in classes on data structures and algorithms, the findings showed that assessment consistency increased and grading time was reduced from weeks to days. The viability of employing PBT as a supplementary tool for extensive automated grading is illustrated by this case study.

In another perspective, Wrenn et al. [Wrenn et al., 2020] discusses the pedagogical aspect of PBT and contends that using relational problems can greatly increase its efficacy in the classroom. For instance, in graph problems where an input may have more than one legitimate output, rendering standard unit tests is insufficient. The authors contend that because these situations force students to think in terms of broader criteria, they are especially well-suited to encouraging the use of PBT. Writing validation predicates and, in certain situations, developing input generators are among the suggested tasks. The study, based on multiple editions of university courses, suggests that students are not only capable of using PBT in these contexts but also value its practical usefulness in preparing for the future job.

In support of this viewpoint, the study conducted by Nelson et al. [Nelson et al., 2021] focuses more specifically on the quality of properties written by students, in an effort to evaluate and enhance their test predicates. The authors first suggested breaking down characteristics into separate sub-properties in order to make fault identification easier and give more lucid feedback. They discovered, however, that this independence made the tests even more insufficient because it made it possible for failures that only happened when several sub-properties failed at once to go unnoticed. In order to produce tailored test sets that are better at identifying complicated mistakes, the article suggests automated solutions. In many instances, these techniques performed better than teachers' manually prepared test suites, while the latter still identified a few distinct errors.

In contrast to typical manual grading, PBT is continuously highlighted for its ability to provide

immediate, detailed, and transparent feedback, improving the learning process. It provides information about performance and complexity and enables students to comprehend which properties their solutions violate. Although using PBT effectively necessitates certain abilities, especially when defining properties and models, research indicates that these abilities can be learned and that the artifacts produced can be used again in the future.

3.5 Summary

This chapter offered a thorough analysis of the state-of-the-art tools, libraries, and algorithms that underpin Property-Based Testing, emphasizing the steps of input generation and shrinking, as well as studies and key findings derived from the use of this testing technique in academic contexts.

Input generation, which automates the construction of a variety of test cases to provide wide coverage of potential scenarios, was the first area of investigation. Targeted input generation, which is intended to handle particular edge cases and boundary conditions, contrasts with random input generation, which stresses efficiency and unpredictability.

The chapter then examined shrinking algorithms, a procedure for condensing unsuccessful test cases into manageable, minimum forms to aid in debugging. There are three main approaches: internal shrinking, which incorporates the simplification process directly into the input generation algorithm, and integrated and external shrinking, which separates simplification from generation. Mechanisms to optimize the shrinking process were discussed, including the creation of shrinking trees and the use of random entropy.

Table 3.7 gives a thorough review of PBT frameworks, related languages, and their input generation and shrinking features.

Table 3.7: Summary of Property-Based Testing Frameworks and Their Features

Framework	**Language**	**Input Generation**	**Shrinking**	**Open Source**
Check-it	Lisp	Random	External	Yes
PrologCheck	Prolog	Random	External	Yes
theft	C	Random	External	Yes
Rapid-Check	C++	Random	External	Yes
PropEr	Erlang	Random and Targeted	External	Yes
QuviQ	Erlang	Random	Integrated	No
QuickCheck	Haskell	Random	External	Yes
Hedgehog	Haskell	Random	Integrated	Yes
falsify	Haskell	Random	Internal	Yes
Hypothesis	Python	Random	Internal	Yes
Schemathesis	Python	Random and Targeted	Internal	Yes
jqwik	Java	Random	External	Yes
rantly	Ruby	Random	External	Yes
jsverify	JavaScript	Random	External	Yes
Fast-Check	JavaScript	Random	External	Yes
CsCheck	C#	Random	External	Yes
ScalaCheck	Scala	Random	External	Yes

Chapter 4

Methodology

In order to answer the proposed research questions, this dissertation follows a methodology that combines theoretical evaluation and practical experimentation. The aim is to identify the most promising algorithms for instance generation and the shrinking process in the context of Property-Based Testing in Prolog, as well as to evaluate the impact of this approach on automatic code correction systems, especially in comparison with unit tests.

4.1 Proposed Solution and Architecture

The implementation and validation of a Property-Based Testing library created especially for SICStus Prolog form the foundation of the suggested approach. This solution emerged after a theoretical evaluation of the key algorithms behind instance generation and counterexample shrinking, focusing on how they could be adapted to the characteristics of logic programming. The selected algorithms were implemented with consideration of Prolog's untyped nature, its reliance on backtracking, and the challenges posed by its non-determinism behaviour. Rather than aiming to create a general-purpose tool, the purpose of this solution is to assess whether PBT techniques can be effectively applied in Prolog.

The PBT library's base architecture is made up of these two elements. In order to compose generators and support user-defined extensions, the first module manages the creation of test inputs from declarative property specifications. The second module is in charge of making test failures easier to understand by shrinking failing cases into minimum counterexamples. Following implementation, the library was linked to one pre-existing automatic grading system that uses only unit testing. Without changing the main objectives of the research, this integration provided a useful means of evaluating the library's capabilities in an actual and real environment.

The architecture's last part offers metrics collecting and instrumentation for comparative analysis. Analyzing performance variations and problem detection capabilities is made possible by executing both property-based and unit tests over a varied collection of student submissions. The solution's modular design guarantees that every stage of development and assessment, including algorithmic adaptation, library implementation, system integration, and comparative analysis, can

be examined while still advancing the main objective, which is to confirm the viability and utility of Property-Based Testing in SICStus Prolog environments.

4.2 Planning and Work Organization

Starting with a theoretical analysis and on to actual implementation, the work was structured to advance step by step, ensuring that each stage added to the overall objective.

4.2.1 Theoretical Evaluation and Selection of Algorithms

To answer the first two research questions, RQ1 and RQ2, a theoretical and exploratory review of the most commonly used algorithms in Property-Based Testing was conducted, focusing on the implementations found in tools such as QuickCheck, Hypothesis, and Falsify. The goal was to identify the most suitable algorithms for instance generation and shrinking in the context of Prolog. The selection of algorithms took into account several particularities, namely:

- algorithms specifically designed for dynamically typed languages;
- algorithms that fully leveraged Prolog's mechanisms, such as unification and backtracking;
- algorithms that offered the flexibility for users to override generator or shrinker behavior, for example, to test more specialized data structures.

Two algorithms were selected as the most promising for adaptation and implementation in Prolog in Section 5.1, aiming to achieve optimal performance in generating diverse test data and minimizing counterexamples in the event of failures, while also striving for a balance between manual effort and the quality of the results.

4.2.2 Experimental Development

After the theoretical selection, the two chosen algorithms were implemented following best practices in logic programming and adhering to the structural characteristics of the language. During implementation, the algorithms were adapted to accommodate peculiarities such as depth-first execution, use of the backtracking mechanism, and inherent non-determinism.

SICStus[1] Prolog was selected as the development framework primarily due to the characteristics outlined in Section 2.2.3. Its industrial-grade reliability, compliance with ISO standards, and proven effectiveness in managing complex systems [Carlsson and Mildner, 2010] made it particularly suitable for implementing and evaluating the selected algorithms. Moreover, a significant factor in this choice was its use at the Faculty of Engineering of the University of Porto (FEUP) in Prolog-related courses, especially in the context of student assessments. This alignment facilitated the intended application of the tool as a support mechanism for testing student submissions

[1] Available online at: `https://SICStus.sics.se/index.html` (last access on June, 2025)

in exams and practical exercises. Version 4.9.0 of SICStus Prolog was chosen because, at the time of the requirements analysis, it was the most recent version of the interpreter available.

All features were then documented with illustrative examples, along with the main limitations and challenges encountered throughout the process of conceptualizing, designing, and developing the framework based on this testing technique.

4.2.3 Evaluation

To answer the third Research Question, RQ3, a case study was conducted on automatic code correction systems in an academic environment to evaluate the developed PBT library. In this study, the PBT tool developed for Prolog, using the best algorithmic approach identified in the previous evaluation step, was integrated into a system for the automatic assessment of programming exercises, which was previously based on unit testing.

The aim was to compare the effectiveness of Property-Based Testing with the traditional Unit Testing approach, based on performance and test quality metrics. Since there is a wide variety of possible exercises, both in terms of complexity and the data structures involved, representative exercises were chosen that reflect a range of academic-level problems. The comparison between PBT and UT included:

- **Execution efficiency**: Measured how long each approach took to generate test case instances and evaluate the submitted code;
- **Fault detection efficiency**: Evaluated the accuracy of fault detection by fixing unit tests and properties on both sides of the testing process and comparing the detection rate for each method. It highlighted which approach better identified faults across different testing scenarios.

During the testing phase, comprehensive instrumentation was used to increase the validity of the comparison. Every student's submission was categorized based on whether it ran well, failed because of syntax errors, or caused runtime problems like timeouts or infinite loops. The evaluation's accuracy and fairness were guaranteed by these classifications, which made it possible to separate out legitimate submissions for the purpose of computing performance and detection measures. Each valid submission's execution time was measured separately, accounting for the time needed for each created property instance and each unit test, as well as any related shrinkage steps. Property shrinking created a predictable overhead, but this feature was crucial in reducing the number of counterexamples and enhancing the clarity of the feedback.

Lastly, the evaluation took each approach's setup complexity into account. Property-based testing benefited from generalization through reusable generators, whereas unit testing frequently necessitated a thorough manual enumeration of edge cases. This trade-off between evaluation coverage and development effort was essential to comprehending each method's suitability for automatic grading systems.

This approach allowed a robust evaluation of algorithms and the application of PBT in SICStus Prolog, both from a theoretical and practical point of view, considering its impact on academic systems and direct comparison with unit tests.

4.3 Risk Assessment

The PBT solution for Prolog in this dissertation could be affected by some problems. Most of these issues are related to the algorithms to be implemented, given the characteristics of this programming language. The main risks are:

- **R1**: There is always a need to implement custom generators and shrinkers for more advanced data structures;
- **R2**: The chosen algorithms are directly constrained by the lack of types in Prolog, requiring the use of more basic algorithms that are less efficient or yield poorer results compared to other approaches in the field;
- **R3**: The ability to handle infinite constructor structures is entirely dependent on the prior implementation of narrowing in Prolog, which adds complexity to the solution;
- **R4**: The PRNG libraries in SICStus Prolog do not allow for the visualization of seed history, which directly prevents the implementation of Internal Shrinking, a key part of the shrinking process in property-based testing;
- **R5**: The input generation algorithm takes too long to process, which can hinder the performance and efficiency of the testing process;
- **R6**: The shrinking algorithm takes too long to process, similarly affecting the efficiency of the overall testing framework;
- **R7**: The case study evaluation is conducted with limited data, as the relevant university course on logic programming is only offered in the first semester, restricting the number of available students and test cases.

Table 4.1 proposes solutions to address these possible issues and categorize the likelihood and their impact in the final solution.

Table 4.1: Possible problems and their solutions

Risk	Likelihood	Impact	Possible Solution
R1	Medium	Medium	Implement default generators and combinators that can be combined with each other. Other untyped languages, such as Erlang and Python, offer PBT libraries with relatively low computational or mental cost when applying this technique [Lampropoulos et al., 2018].
R2	High	Medium	None. There are specific algorithms that directly benefit from the characteristics of the programming languages involved.
R3	Medium	Medium	Narrowing in Prolog [Cheong and Fribourg, 1998] has already been explored and facilitates handling structures and constructors of this type.
R4	Medium	High	Opt for implementing external or integrated shrinking as a subset of the properties of internal shrinking, as this approach does not explicitly depend on the internal functioning of PRNG [de Vries, 2023].
R5	Medium	Medium	The use of constraint programming could improve the time efficiency of some of the algorithms for input generation [Blanco et al., 2019].
R6	Medium	Medium	The use of constraint programming could improve the time efficiency of some of the algorithms for shrinking [Blanco et al., 2019].
R7	Medium	High	Focus the study on the files used in previous years of the same university course.

4.4 Summary

This chapter outlines the methodological approach adopted in this dissertation, structured around theoretical analysis, practical development, and empirical evaluation. The best algorithms for instance generation and shrinking in Prolog were identified through an evaluation of existing Property-Based Testing tools. These algorithms were selected for their adaptability to the declarative and dynamic characteristics of the Prolog programming language. Their implementation was carried out in SICStus Prolog, a choice justified by the tool's relevance in educational settings and compliance with ISO standards. The development process also considered specific aspects of logic programming, such as backtracking, non-determinism, and the absence of static types.

A case study within an academic automatic code correction system serves to assess the practicality of the proposed approach. The evaluation compares Property-Based Testing with conventional Unit Testing using metrics such as execution time, fault detection accuracy, and test setup complexity. Instrumentation supports detailed data collection on test coverage, error types, and performance, providing a nuanced perspective on the contexts in which each testing strategy proves most effective. The chapter also discusses potential risks related to algorithmic limitations and contextual constraints, along with the mitigation strategies considered to reinforce the robustness of the proposed solution.

Chapter 5

Design and Development of a PBT Library for SICStus Prolog

This chapter discusses the design and implementation of a Property-Based Testing library specifically for SICStus Prolog, taking into account the need to identify the algorithms best suited to the language and environment. The development was guided by an analysis of how well different algorithms align with Prolog's unique characteristics, supported by the risk assessments outlined in the previous chapter and the feasibility of the solutions explored throughout the process.

5.1 Algorithms Selection

The algorithms that create test cases and shrink counterexamples must be carefully chosen to facilitate the creation of a Property-Based Testing library that is appropriate for Prolog.

The algorithms were selected with a view to using this framework with the SICStus[1] interpreter. SICStus Prolog was selected as the development framework primarily due to the characteristics outlined in Section 2.2.3. Its industrial-grade reliability, compliance with ISO standards, and proven effectiveness in managing complex systems [Carlsson and Mildner, 2010] made it particularly suitable for implementing and evaluating the selected algorithms. Moreover, a significant factor in this choice was its use at the Faculty of Engineering of the University of Porto in Prolog-related courses, especially in the context of student assessments. This alignment facilitated the intended application of the tool as a support mechanism for testing student submissions in exams and practical exercises. Version 4.9.0 of SICStus Prolog was chosen because, at the time of the requirements analysis, it was the most recent version of the interpreter available.

The justification for the algorithms chosen for implementation is provided in this section. The objective is to find approaches that guarantee flexibility, performance, and user extensibility while simultaneously being theoretically sound and compatible with Prolog's operational semantics. Using the theoretical examination from the previous chapter as a guide, the selection procedure selects the options that show the greatest promise for being incorporated into the logical paradigm.

[1] Available online at: https://SICStus.sics.se/index.html (last access on June, 2025)

5.1.1 Input Generation

Random input generation and targeted input generation are the two main categories into which PBT's input generation algorithms can be broadly divided. Targeted generation seeks to direct the input space research toward certain, frequently more troublesome locations, whereas random generation concentrates on broad coverage through unpredictability. These approaches vary in how they investigate the input space of a given function or attribute.

Random input generation is present in all the libraries examined in the state-of-the-art analysis in Chapter 3, regardless of the programming language used or its type system, including both statically and dynamically typed languages. Prolog is a dynamically typed language, so the implementation of a random input generation algorithm is not only feasible but also aligns with existing practices in comparable environments such as Python and Erlang.

One consistent requirement across all PBT libraries is the need for a pseudo-random number generator (PRNG). SICStus Prolog provides built-in support for random number generation through the random[2] library, which offers predicates such as *random/1* and *setrand/1*. To ensure that generated values are truly non-deterministic, seeds must be initialized properly, commonly using system time or user input. Furthermore, a shared characteristic across existing libraries is the ability to build complex generators by composing simpler ones. This composability makes it easy to define generators for domain-specific and structured data types. The Prolog library will ensure that Risk R1, the constant need to implement custom generators and shrinkers for more advanced data structures, is mitigated by offering combinable and reusable generator patterns.

Similarly, Risk R3, the challenge of handling infinite constructor structures, requires using Prolog's narrowing feature, which greatly raises the solution's complexity [Cheong and Fribourg, 1998]. The method used here chooses not to use narrowing in order to avoid this extra complexity and any potential performance issues. In order to avoid indefinite loops or infinitely deep structures, the emphasis is instead on ensuring that the data production process will always finish. It provides a more straightforward and effective option, despite the fact that guaranteeing termination can be challenging. Listing 3.4, which is based on the work of Lampropoulos et al. [Lampropoulos et al., 2018], provides an example of this tactic. This termination limit can be parameterized and used as a means to introduce entropy into the structures generated through random generation

Targeted input generation, as explored in Chapter 3, is predominantly implemented in dynamically typed languages such as Python and Erlang. The algorithms typically associated with this approach, such as Hill Climbing and Simulated Annealing, are grounded in the concept of a search strategy, which requires two fundamental components: a definition of a utility value to evaluate the quality of each test case and an understanding of the data structure being manipulated. These techniques frequently build regions and optimization routes using implicit or inferred type knowledge. Additionally, they require the tester to possess domain-specific expertise in order to define suitable utility functions. It is the user's responsibility to define the context or goal of a given property because it cannot be deduced from its input arguments alone. Simplifying the problem from

[2]Available online at: https://SICStus.sics.se/SICStus/docs/4.3.1/html/SICStus/lib_002drandom.html (last access on June, 2025)

a high-level perspective, implementing full-fledged targeted input generation with explicit utility functions and neighborhood exploration is unnecessarily complex in the context of Prolog. [Amaral et al., 2014] in the context of PrologCheck proposed a frequency parameter that enables users to assign probabilities to different generators. With the help of this method, testers can successfully imitate intended behavior without requiring full optimization infrastructure by influencing the likelihood that specific situations will be generated. Allowing users to direct the generator through configuration allows for the prioritization of edge cases or input patterns, like empty lists or boundary values that are known to reveal errors. This method extends random generation and is implemented in the library as an optional yet user-friendly feature. Generator composability is maintained, indicating that Risk R1 is being successfully reduced in real-world scenarios.

The adopted approach for the library will prioritize random input generation due to its broad applicability and alignment with Prolog's characteristics. However, the design will not preclude the incorporation of targeted input generation techniques, particularly those that rely on the tester's domain knowledge to guide input generation through enhanced configuration mechanisms. On top of fundamental generators, users could create custom generators that are suited to edge cases or troublesome input patterns, thanks to this flexibility.

5.1.2 Shrinking

Shrinking techniques in Property-Based Testing can be classified into three categories, listed in increasing order of complexity and innovation: external shrinking, integrated shrinking, and internal shrinking. Each approach presents different trade-offs in terms of control, expressiveness, performance, and integration with the input generation mechanism.

Risk R4, identified in Chapter 4, explains why implementing internal shrinking is currently infeasible in SICStus Prolog. Unlike libraries such as Falsify for Haskell or Hypothesis and Schemathesis for Python, which support internal shrinking by capturing and replaying the pseudo-random seed history, SICStus Prolog's pseudo-random number generators libraries do not provide any interface for inspecting or reproducing the sequence of generated random values. In order for internal shrinking to function properly, it is essential that no attempt be made to mechanically reproduce a failing test case by replicating the exact same generation sequence from a particular seed. This limitation does not stem from any intrinsic characteristics of the Prolog language itself, unlike what was considered in Risk R2, but rather from the specific constraints of the SICStus Prolog runtime and its PRNG implementation. The choice to exclude internal shrinking is technical and tool-specific, not conceptual. However, this opens up the possibility for the future development of a PRNG library with these characteristics, paving the way for reconsidering and potentially reselecting the underlying algorithm.

Integrated shrinking attempts to unify input generation and shrinking by representing generated values as tree structures, where the root is the initial input and each child represents a progressively smaller or simpler version of it. This methodology simplifies accuracy and frequently increases efficiency by guaranteeing that the shrinking process complies with the same structural invariants and restrictions imposed by the generator. From a theoretical perspective, the method

is especially appealing because it lowers the possibility of breaking generation invariants when downsizing. Libraries such as QuviQ in Erlang and Hedgehog in Haskell successfully implement integrated shrinking. However, there are structural and performance costs associated with this method, particularly when applying it to a language like Prolog that does not have a robust static type system. SICStus Prolog lacks facilities for type inference, which is useful in statically typed languages to ensure proper shrink routes and generator structures. Therefore, although Risk R2, the limitation imposed by the lack of types in Prolog, which forces the use of simpler, less efficient algorithms compared to other approaches, does not directly block the use of integrated shrinking, the additional complexity and lack of supporting abstractions in Prolog make this option impractical for the current implementation. Integrated shrinking is tightly coupled with the generator logic, often requiring users to think in terms of shrink trees and recursive invariants. Considering these factors, integrated shrinking was excluded from the design of the proposed library in favor of more flexible and simpler alternatives.

All PBT libraries, regardless of the programming language, use external shrinking algorithms. For every data type, it entails designing explicit shrinking functions that run apart from the generators. The implementation of this approach is not hindered by the peculiarities of Prolog because it relies on functional decomposition instead of type inference or runtime inspection. Notably, Prolog's built-in backtracking mechanism can help with the investigation of shrink paths since it allows for the spontaneous production and testing of different shrink candidates without the need for explicit state management or conventional recursion.

The external shrinking approach offers significant flexibility and fine-grained control but comes with its own challenges. As noted by Pike [Pike, 2014] and Chen [Chen et al., 2022], maintaining alignment between generators and shrinkers requires careful attention. Shrinkers must preserve the same invariants upheld by their corresponding generators, otherwise they risk introducing invalid inputs or irrelevant test cases [de Vries, 2023]. Given that complex data types increase the need for bespoke shrinkers, this supports the applicability of Risk R1. Inspired by solutions from untyped languages like Python and Erlang, which have demonstrated that the computational and cognitive weight can be kept low provided appropriate abstractions are in place, the proposed library will include a collection of default shrinkers and composable shrinker combinators to lessen this. Nevertheless, it is acknowledged that redundancy may arise due to the similar logic often required for both generation and shrinking.

In light of these factors, external shrinking was determined to be the best approach for the PBT library's initial implementation in SICStus Prolog. It provides the optimum balance between practicality, adaptability, and compatibility with Prolog's features, but it also necessitates greater user discipline and effort when developing shrinkers for unique data types. Since this algorithm permits reusable combinators and straightforward yet expressive shrinking functions, even in the absence of a static type system, risks R1 and R2 were carefully evaluated and directly influenced the choice, and were mitigated.

5.2 Development Environment

SICStus Prolog was chosen as the development framework due to its industrial-grade reliability, adherence to the ISO standard, and proven effectiveness in handling complex systems [Carlsson and Mildner, 2010]. Additionally, it was selected because the planned case study for evaluating the library is based on data from the Functional and Logic Programming (PFL) course, which uses this interpreter as part of its curriculum.

There is currently limited investment in Property-Based Testing libraries for the Prolog language, regardless of the environment or interpreter used. However, two candidate libraries were identified as potential foundations for development: PrologCheck [Amaral et al., 2014] and Prolog QuickCheck[3]. Both are considered deprecated, having seen no updates for over six years, and neither was originally developed for SICStus Prolog.

The decision to adopt Prolog QuickCheck as a base was driven by three factors: it is open source; despite its lack of recent maintenance, it was developed in SWI-Prolog, which shares a substantial degree of compatibility with SICStus Prolog; and it includes an initial implementation of both random input generation and external shrinking, two core algorithmic strategies selected and justified in the previous section.

The first step in the implementation involved adapting the original source code to ensure compatibility with SICStus Prolog. Given the syntactic similarity between SWI-Prolog and SICStus Prolog, most of the code required only minor adjustments, particularly with regard to library dependencies. Whenever possible, built-in SICStus libraries were used to maximize compatibility with future versions of the interpreter.

The features presented in the following sections result from an extension of the original library, involving significant adaptation, expansion, and enhancement efforts.

5.3 Support for Basic and Composite Data Types

The developed framework supports the generation of both basic and composite data. Standard data types like integers and booleans are examples of basic types, and sets are examples of composite data structures. User-declared transformation rules that govern how more complex structures should be formed from simpler components can be used to define composite types.

The `arbitrary/2` predicate supports fundamental types, as shown in Listing 5.1. For most situations, these definitions offer regulated random generation and may also include calls to auxiliary predicates to support more complex value construction. This example explicitly demonstrates the generation of booleans, signed 32-bit integers, and sets of integers.

```
1 arbitrary(boolean, X) :-
2     random_member(X, [true, false]).
3
```

[3] Available online at: https://github.com/nicoabie/quickcheck (last access on June, 2025)

```
arbitrary(integer, X) :-
    random_between(-2147483648, 2147483647, X).

random_int(X):- arbitrary(integer, X).
arbitrary(integer_set, Set) :-
    random_between(0, 1000, Length),
    length(X, Length),
    maplist(random_int, X),
    remove_duplicates(X, Set).
```

Listing 5.1: Example of arbitrary generators for basic types

Composite types are defined using the `composite/3` predicate, which specifies how to build a value of a given type from one or more base types. To ensure type safety, a `has_type/2` predicate must also be defined for each composite type. This serves two purposes: it compensates for Prolog's dynamic type system by allowing type checks only when necessary, and it supports more controlled and meaningful manipulation of variables throughout the program. This predicate is acting as a runtime validator that ensures values respect both the structure and invariants of their types. In Listing 5.2, a generator produces even integers and implicitly enforces an invariant that must be preserved.

```
composite(even, [S:integer], X) :- X is 2 * S.

has_type(even, X) :-
    has_type(integer, X),
    0 is X mod 2.
```

Listing 5.2: Definition of composite types for even integers with type validation

However, without a validator, other transformations or shrinkers may silently produce invalid values that violate this constraint. By explicitly encoding invariants, it ensures that both generated and shrunk values remain valid instances of the intended type, improving the reliability of tests and reducing duplicated logic in custom shrinkers.

As explained in Section 5.1, compositional generators provide for a more straightforward method of targeted input generation that is tailored to Prolog's characteristics. A generator can be made to bias the generation towards these ranges for a particular test scenario where the tester is already aware that the function being tested involves edge cases involving, for example, negative integers or values near zero. This idea was inspired by the work developed in PrologCheck, specifically in the composition of generators through the explicit declaration of probabilities [Amaral et al., 2014]. This preserves generality while enabling the tester to direct input generation. Listing 5.3 demonstrates the library's ability to enable this type of focused generation.

```
arbitrary(small_negative, X)
    :- random_between(-100, -1, N).
```

```
composite(targeted_number, [SN:small_negative, I:integer], X) :-
    random(R),
    ( R < 0.8 -> X = SN      % 80% chance
    ; X = I                  % 20% chance
    ).
```

Listing 5.3: Targeted input generation using compositional generators

Similarly, it is also possible to create more elaborate types, such as compound terms that handle multiple types. The terms themselves are also composable, as demonstrated in Listing 5.4.

```
composite(person, [Name:string, Age:positive_integer], person(Name, Age)).

has_type(person, person(Name, Age)) :-
    is_of_type(string, Name),
    is_of_type(positive_integer, Age).

composite(course, [University:string, Professor:person],
    course(University, Professor)).

has_type(course, course(University, Professor)) :-
    is_of_type(string, University),
    is_of_type(person, Professor).
```

Listing 5.4: Complex composite types using multiple base values

The listings above are supported by the `generate_argument/2` predicate, responsible for transforming type annotations into concrete values. This predicate supports both the generation of basic types and the recursive handling of composite type structures, as illustrated in Listing 5.5.

```
% simple arbitrary types
generate_argument(_:Type, [Value:Type]) :-
    arbitrary(Type, Value).

% composites of one arbitrary type
generate_argument(_:Type, [Value:Type, BaseValue:BaseType]) :-
    clause(composite(Type, _:BaseType, _), _),
    generate_argument(_:BaseType, [BaseValue:BaseType]),
    composite(Type, BaseValue:BaseType, Value).

% composites of many arbitrary types
generate_argument(_:Type, [Value:Type|BaseValues]) :-
    clause(composite(Type, [HBaseTypes|TBaseTypes], _), _),
    generate_arguments([HBaseTypes|TBaseTypes], BaseValues, _),
    composite(Type, BaseValues, Value).

```

```
generate_arguments(Args, Values, ValuesWithBaseTypes) :-
    maplist(generate_argument, Args, ValuesWithBaseTypes),
    maplist(head, ValuesWithBaseTypes, Values).
```

Listing 5.5: Logic for generating arguments from base and composite types

This logic ensures that any type, basic or composite, can be correctly interpreted and instantiated before being passed to a property under test. The system is resilient even without static typing thanks to this technique, which offers flexibility, type-checking at runtime and random or targeted test case generations.

5.4 Shrinking Capabilities

The library shrinks the counterexample to create a smaller one when a property fails, methodically reducing the corresponding input to its most basic form and identifying minimal, repeatable test cases. Listing 5.6 describes the general algorithm used in the library.

```
shrink_example(Depth0, Module, Property, ValuesWithBaseTypes, Example) :-
    Depth0 < 32,
    shrink_arguments(ValuesWithBaseTypes, Shrunk, ShrunkWithBaseTypes),
    ValuesWithBaseTypes \== ShrunkWithBaseTypes,
    ShrinkGoal =.. [Property|Shrunk],
    \+ Module:ShrinkGoal, !,
    Depth is Depth0 + 1,
    shrink_example(Depth, Module, Property, ShrunkWithBaseTypes, Example).

shrink_example(Depth,_,_,ValuesWithBaseTypes, Example) :-
    warn('Shrinking to depth ~d', [Depth]),
    maplist(head, ValuesWithBaseTypes, Example).
```

Listing 5.6: Recursive shrinking of counterexamples to simpler failing cases

As observed, shrinking has a finite depth to prevent infinite loops, since the initially found counterexample may already be minimal. On the other hand, because the implemented shrinking mechanism is external, the user must explicitly declare all shrinkers for every custom type, both simple and compound. Listing 5.7 presents three concrete examples supported by the library.

```
shrink(odd, 1, 1) :- !.
shrink(odd, Odd, ShrunkValue) :-
    ShrunkValue is Odd - 2.

shrink(tuple(integer, string), (I, S), (ShrunkInteger, ShrunkString)) :-
    shrink(integer, I, ShrunkInteger),
    shrink(string, S, ShrunkString).

```

```
shrink(short_int, _, 0).
shrink(short_int, X, Y) :-
    X < 0,
    Y is -X.
shrink(short_int, X, Y) :-
    X > 0,
    Y is X // 2.
```

Listing 5.7: Examples of user-defined shrinkers for simple and compound custom types

As demonstrated by the shrinker for odd integers, the user is responsible for preserving the defined invariants, which is one of the main drawbacks of external shrinking. This problem is mitigated by runtime type checking, which is made possible by the previous validation covered in subsection 5.3.

Composition is also powerful in this context. Since composite types are built from the composition of other generators, their shrinkers are simply the result of invoking the shrinkers of the types they are composed of. Similarly, by understanding the problem context and its most likely edge cases, it is possible to design targeted shrinkers, as demonstrated in the example, in order to reduce the shrink depth and the overall execution time of this stage in Property-Based Testing.

5.5 Knowledge Base Interaction

Prolog's support for logic facts and dynamic knowledge databases is one of its unique features. In order to include this characteristic with Property-Based Testing, the library adds a specific layer for using generated data, both simple and composite, to interface with the Prolog database, as discussed in the previous section. This feature enables properties to populate the knowledge base dynamically and validate behaviors that depend on database state.

The internal implementation of types is separated from the development of databases, as shown in Listing 5.8.

```
% create_database(+Type)
% Generates 100 values of the given Type and stores them.
create_database(Type) :-
    create_database(Type, 100).

% create_database(+Type, +Number)
% Generates Number values of the given Type and stores them.
create_database(_, 0) :- !.
create_database(Type, Number) :-
    generate_composite(Type, X),
    assertz(X),
    NextNumber is Number - 1,
    create_database(Type, NextNumber).
```

```
% generate_composite(+Type, -Value)
% Generates a single composite value of the given Type.
generate_composite(Type, X) :-
    composite(Type, Args, _),
    generate_arguments(Args, Values, _),
    composite(Type, Values, X).
```

Listing 5.8: Recursive creation of generated composite terms in the dynamic database

One can build a custom count as needed or a default number of facts by using generic constructors like *create_database/1* and *create_database/2*. This abstraction improves modularity and reusability by insulating the test logic from the specifics of a type's generation.

A simplified store using `assertz` directly for all facts generated during this process is indeed necessary. Initially, for greater modularity, the component responsible for interacting with the knowledge base was designed to shield the tester from the library's internal implementation details, including the arity of each fact. This allowed the tester to simply call predicates like `get_from_database(course)` or `clear_database(course)` to retrieve or remove all `course` facts from the fact base. However, SICStus Prolog required the explicit specification of arity. To address this limitation, the `store_term` predicate extracted the functor of the given term and stored it as a key for a generic `stored` value, as detailed in Listing 5.9.

```
% store_term(+Value)
% Stores the value, associating it with its functor name as the Type.
store_term(X) :-
    functor(X, Type, _),
    assertz(stored(Type, X)).
```

Listing 5.9: Initial approach for storage generated facts

Retrieving facts is implemented via *get_from_database/2*, which returns all values of a given type as a list. This feature is especially useful for asserting the internal state of the database, for example, to confirm whether the correct number of facts remain after deletions or manipulations performed during the test. Additionally, *get_from_database/1* retrieves all entries across all types, simplifying total state validation in tests involving multiple entities. The solution with *store_term(X)* initially appeared to work well, with the internal implementation of the predicates defined above shown in Listing 5.10.

```
% get_from_database(+Type, -List)
% Retrieves all values of the given Type from the database.
get_from_database(Type, List) :-
    findall(X, stored(Type, X), List).

% clear_database(+Type)
% Removes all stored values of a specific Type.
```

```
clear_database(Type) :-
    retractall(stored(Type, _)).
```

Listing 5.10: Retrieval and cleanup of generated facts

However, with the constant reliance on a general *stored* fact, the code under test within the properties would not be able to access the facts as they exist in the fact base. In other words, any predicate that PBT aimed to test and that manipulated dynamically generated facts in the database would have to be adapted, since these facts would not be directly accessible and there is no explicit mention of the internal use of *stored*. Therefore, the decision was made to store the generated facts exactly as they are using *assertz*, and in cases where consulting or removing facts from the fact base requires explicit arity, this is handled as shown in Listing 5.11.

```
% get_from_database(+Type/Arity, -List)
% Retrieves all facts with functor Type and arity Arity into List
get_from_database(Type/Arity, List) :-
    functor(Template, Type, Arity),
    findall(Template, Template, List).

% clear_database(+Type/Arity)
% Removes all facts with functor Type and Arity from the knowledge base
clear_database(Type/Arity) :-
    functor(Template, Type, Arity),
    retractall(Template).
```

Listing 5.11: Retrieval and cleanup of generated facts with explicity arity

The library does not lose modularity, as the generation of facts requires the tester to explicitly specify both the arity and the content of the fact. For instance, Listing 5.12 demonstrates how this interface allows for the independent instantiation of known person and course entities.

```
prop_create_courses(Count:positive_integer) :-
    create_database(course, Count),
    get_from_database(course/3, Courses),
    length(Courses, Count),
    clear_database(course/3).

prop_get_database(PersonCount:positive_integer, CourseCount:
    positive_integer) :-
    create_database(person, PersonCount),
    create_database(course, CourseCount),
    get_from_database(All),
    length(All, Total),
    Total is PersonCount + CourseCount, clear_database.
```

Listing 5.12: Properties for validating database population and global retrieval

The library offers database cleaning tools to preserve test independence and provide predictable results. As seen in Listing 5.13, one can use *clear_database/0*, *clear_database/1*, or *clear_database/2*, respectively, to clear all data, a certain kind, or a restricted number of facts from a specified type.

```
prop_clear_type(PersonCount:positive_integer, CourseCount:positive_integer
    ) :-
    create_database(person, PersonCount),
    create_database(course, CourseCount),
    clear_database(person),
    get_from_database(person, Ps), length(Ps, 0),
    get_from_database(course, Cs), length(Cs, CourseCount),
    clear_database.

prop_clear_limited(PersonCount:short_int, ToRemove:short_int) :-
    Max is max(1, PersonCount),
    Rm is min(Max, ToRemove),
    create_database(person, Max),
    clear_database(person, Rm),
    get_from_database(person, RemainingPersons),
    length(RemainingPersons, Remaining),
    Remaining is Max - Rm,
    clear_database.
```

Listing 5.13: Selective clearing of database entries by type to ensure isolation

It is important to note that fact generation is fully independent: each fact is generated in isolation, without dependencies on other existing entries. As explained in Section 5.7, this makes generation easier but restricts support for related data in more complex problems.

This feature of the library developed for PBT enables the testing of properties and predicates that interact with knowledge databases in various ways, offering flexibility in both the structure and content of the generated facts. No incompatibilities were identified between the use of these dynamic facts and the other features discussed in this chapter.

5.6 Testing Interface Overview

The fundamental elements of the user interface are described in depth in the following subsections, which also show how properties are defined, used, and reported within the framework. This overview emphasizes the fundamental techniques and usability features that provide efficient Property-Based Testing.

5.6.1 Property Execution and Test Generation

In the developed library, a property is defined as any predicate that takes zero or more arguments, each accompanied by its corresponding type. In Listing 5.14, an example is shown where a typed argument is used to define a property that verifies the behavior of a binary conversion predicate.

```
% Tests for cases [1, 0, 0, 0, 0, 0, ...]
prop_dec2bin_power(Size:integer) :-
        Value is 2 ^ (Size - 1),
        dec2bin(Value, [1 | Tail], Size),
        maplist(=(0), Tail).
```

Listing 5.14: Property definition illustrating the use of typed input in the PBT framework

The explicit specification of each argument's type using the *:* metapredicate is mandatory. This approach serves multiple purposes: it improves code readability for the user, provides visual feedback that aids debugging, and enables tracking of data types within Prolog. As a result, it ensures that the appropriate generator and shrinker are correctly invoked for each argument. This design choice also facilitates the automatic association between types and their corresponding testing logic, streamlining the test execution process. Within the body of the property, the predicate under test must be called at some point. Additionally, auxiliary predicates or external library calls may be used freely to support the validation logic. The property holds if the body succeeds.

From a general perspective, properties are executed by invoking the *quickcheck* predicate with the name, following the design inspiration of QuickCheck in Haskell, plus the arity of the property to be tested, as shown in Listing 5.15.

```
?- quickcheck(prop_dec2bin_power/1).
100 tests OK
```

Listing 5.15: Example of executing a property with specified arity using the quickcheck predicate

Multiple properties with the same name but different arities are supported, allowing for greater modularity and flexibility. Additionally, it is possible to override the default maximum number of test case generations, which is set to 100, by specifying a custom limit, providing further control over the testing process, as presented in Listing 5.16.

```
quickcheck(Module:Property/Arity) :-
    quickcheck(Module:Property/Arity, 100).

?- quickcheck(prop_dec2bin_power/1, 1500).
1500 tests OK
```

Listing 5.16: Quickcheck with configurable test count and arity handling

The overall mechanism is illustrated in Listing 5.17. Internally, the *quickcheck/2* predicate initializes the random seed to ensure variability in test case generation, as discussed in the algorithm selection section 5.1. This initialization uses SICStus Prolog's built-in *random*[4] and *system*[5] libraries, taking advantage of the current runtime statistics to seed the pseudo-random number generator. This process helps guarantee that generated values differ across test executions.

```
:- use_module(library(random)).
:- use_module(library(system)).

init_random_seed :-
    statistics(runtime, [S | _]),
    setrand(S).
quickcheck(Module:Property/Arity, TestCount) :-
    init_random_seed,
    functor(Head, Property, Arity),
    once(Module:clause(Head, _)),
    run_tests(TestCount, Module, Property, Args, Result),
    handle_result(Result, TestCount, Property).

handle_result(ok, TestCount, _) :-
    warn('~d tests OK', [TestCount]), !.

handle_result(Example, _, Property) :-
    ExampleGoal =.. [Property | Example],
    warn('Failed test ~q', [ExampleGoal]).
```

Listing 5.17: Quickcheck predicate internal logic with seed initialization and test execution

The *run_tests/5* predicate manages the main testing logic, as presented in Listing 5.18, which includes creating test inputs, running the property being tested, and gathering the outcomes. It works with the shrinking and generating mechanisms that are specified elsewhere in the library. This abstraction enhances the framework's flexibility and maintainability by isolating the testing flow from generator logic and property declarations, allowing for a clear separation of duties.

```
run_tests(0, _, _, _, ok).

run_tests(TestCount, Module, Property, Args, Example) :-
    TestCount > 0,
    generate_arguments(Args, Values, ValuesWithBaseTypes),
    Goal =.. [Property | Values],
```

[4]Available online at: https://sicstus.sics.se/sicstus/docs/4.3.1/html/sicstus/lib_002drandom.html (last access on June, 2025)

[5]Available online at: https://sicstus.sics.se/sicstus/docs/4.3.1/html/sicstus/lib_002dsystem.html (last access on June, 2025)

```
7       handle_test_result(Module, Goal, TestCount, Property, Args,
            ValuesWithBaseTypes, Example).
8
9   handle_test_result(Module, Goal, TestCount, Property, Args, _, Example) :-
10      Module:Goal,
11      Remaining is TestCount - 1,
12      run_tests(Remaining, Module, Property, Args, Example).
13
14  handle_test_result(Module, Goal, _, Property, _, ValuesWithBaseTypes,
        Example) :-
15      \+ Module:Goal,
16      shrink_example(0, Module, Property, ValuesWithBaseTypes, Example).
```

Listing 5.18: Recursive execution of test cases and example shrinking on failure

The generation of input values relies on the internal predicate `generate_argument/2`, which is responsible for producing values of both basic and composite types. In the following subsection, a number of type generator examples and their composition techniques are presented, along with a detailed explanation of this capability.

5.6.2 Meta-Properties

In contrast to paradigms such as functional programming, which often produce only one result, Prolog can deliver several solutions through backtracking, as explained in Section 2.2.3. This is helpful in situations when the objective is to gather all legitimate cases that meet a specific criterion or to determine whether solutions exist.

To test such properties, the library provides a meta-property that checks how many solutions a predicate produces, as shown in Listing 5.19. In this example, the property verifies that exactly one solution exists. This allows developers to easily write properties that test exact solution numbers or possible ranges.

```
1   :- meta_predicate solutions_in_range(+, +, 0, -).
2
3   solutions_in_range(Min, Max, Generator, Template) :-
4       findall(Template, Generator, Results),
5       length(Results, Count),
6       Count >= Min, (Max = -1 ; Count =< Max).
7
8   prop_dec2bin_one_solution(Size:integer) :-
9       get_random_number(Size, Number),
10      solutions_in_range(1, 1, dec2bin(Number, List, Size), List).
```

Listing 5.19: Checking the number of solutions

Another feature of Prolog is its flexibility regarding argument instantiation. Unlike other paradigms, such as functional programming, where input and output relationships are strictly

fixed, Prolog allows arguments to be instantiated or uninstantiated. This enables different styles of querying and makes Prolog particularly versatile when exploring logical patterns and relations.

As shown in Listing 5.20, the library provides meta-properties to verify whether a variable is instantiated or not. These checks can be placed before or after the predicate under test, allowing for fine-grained control over the testing logic. As can be observed, a renaming strategy was adopted to add a layer of meaning to each step whenever it is included in a property, making its purpose easier to understand. This module represents a part of the library that can be further expanded according to future needs.

```
1 is_instantiated(Var) :- nonvar(Var).
2
3 is_uninstantiated(Var) :- var(Var).
4
5 prop_dec2bin_instantiated(Size:short_int) :-
6     get_random_number(Size, Decimal),
7     dec2bin(Decimal, List, Size),
8     is_instantiated(List).
```

Listing 5.20: Checking if argument is instantiated

5.6.3 Counterexample Reporting

The counterexample report is one of the most relevant features of Property-Based Testing, as it provides direct feedback to the user about the method arguments that falsify a given property. However, the implementation of this library, as presented in the previous subsections, does not meet this requirement. Taking the *prop_dec2bin_power* property as an example again, the output when the property fails would be as shown in Listing 5.21.

```
1 prop_dec2bin_power(Size:integer) :-
2     Value is 2 ^ (Size - 1),
3     dec2bin(Value, [1 | Tail], Size),
4     maplist(=(0), Tail).
5
6 ?- quickcheck(prop_dec2bin_power/1).
7 Failed test prop_dec2bin_power(Size:11)
```

Listing 5.21: Failure example for the *prop_dec2bin_power* property

While the first argument of *dec2bin* is the same as that of the property, and the second argument is calculated internally, also serving as input, and the framework can only return to the user the input of the property itself. It was necessary to create a system that allows providing complete feedback because that is the feedback that matters to the user, and often they do not have access to the properties or their implementations used to test their predicate.

Indeed, the inputs of the property do not always match the inputs required by the method under test. One way to work around this situation is to create inputs that depend internally on each other, as exemplified in Listing 5.22, and that fit the inputs needed by the method under test.

```
prop_dec2bin_power(Tuple:tuple) :-
    Tuple = (Size, Value),
    dec2bin(Value, [1 | Tail], Size),
    maplist(=(0), Tail).

?- quickcheck(prop_dec2bin_power/1).
Failed test prop_dec2bin_power(tuple:(11, 1024))
```

Listing 5.22: Property using correlated tuple inputs

In this case, the user should rely on a compound generator that returns two correlated numbers, avoiding performing calculations inside the property. This way, the user has access to the necessary inputs to falsify the property. However, this requires creating generators very specific to properties and each test, and they are probably not reusable for other scenarios. To make the library as modular and oriented to a wider range of problems as possible, the solution was to store the inputs in the Prolog fact base before calling the method under test.

The call to the input method is optional for each property, but if invoked, it substantially improves the message given to the user without knowing in advance the internal implementation of the property. Listing 5.23 illustrates a possible output of this feature.

```
prop_dec2bin_power(Size:integer) :-
    Value is 2 ^ (Size - 1),
    input([value:Value, size:Size]),
    dec2bin(Value, [1 | Tail], Size),
    maplist(=(0), Tail).

?- quickcheck(prop_dec2bin_power/1).
Failed test prop_dec2bin_power with [value:1024, size:11]
```

Listing 5.23: Example of storing inputs as facts for enhanced feedback

The developed library ensures proper support for Property-Based Testing in SICStus Prolog, covering property execution, random input generation, external shrinking, knowledge base interaction, and counterexample reporting. Therefore, it addresses most of the testing needs commonly found in other PBT libraries. Fig. 5.1 illustrates the high-level architecture of the library, as well as the modules involved in the process.

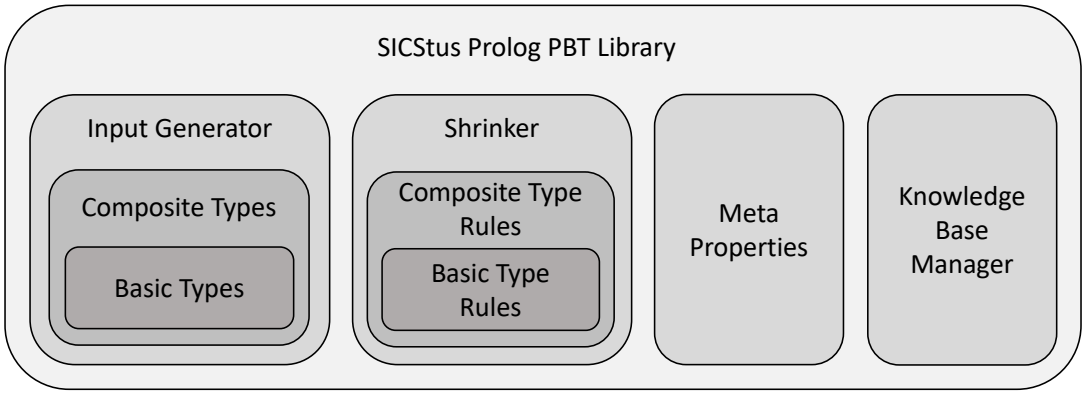

Figure 5.1: Modular Architecture of the developed SICStus Prolog PBT library

5.7 Limitations

Although the implemented library is as generic and reusable as possible for most Prolog testing scenarios, two main limitations of the implementation have been identified: handling tests that involve input/output and handling tests that require a fact base with more complex interconnections.

The library is not adapted to use Property-Based Testing with user input or output. When predicates under test interact with the console, such as reading user input or printing output, the current framework cannot capture or simulate these interactions, which prevents automated testing of such predicates. For example, in Listing 5.24, a predicate that reads a number from the console and performs some calculation based on it cannot be tested with the current implementation since the input must be provided manually, and the output is sent directly to the console without being intercepted by the framework.

```
1 double_input(Result) :-
2     read(Number),
3     Result is Number * 2.
```

Listing 5.24: Predicate reading from input and doubling the number

A possible approach is to simulate the input stream by temporarily redirecting the user input stream to a custom input source containing the desired test value. SICStus Prolog provides built-in predicates that allow I/O stream manipulation. The property can then invoke the predicate normally, reading from the simulated input stream, and verify the result matches expectations, as illustrated in Listing 5.25.

```
1 with_input(Stream, Goal) :-
2     setup_call_cleanup(
3         ( current_input(Old), set_input(Stream) ),
4         Goal,
```

```
        ( set_input(Old), close(Stream) )
    ).

prop_double_input(Input) :-
    format(string(S), '~w.', [Input]),
    open_string(S, Stream),
    with_input(Stream, double_input(Result)),
    Result =:= Input * 2.
```

Listing 5.25: Property test injecting input and capturing output

However, this strategy is overly verbose and requires adjustments in the upper layer that calls the property in order to become modular. It should not be the tester's responsibility to explicitly declare all the steps in each property to handle and test I/O interactions.

The current implementation of the library is capable of constructing, manipulating, storing, and retrieving facts of any format, whether simple or composed of multiple other data types. However, the generation of each fact is done at runtime and independently of the facts generated previously. Listing 5.26 presents a set of interrelated facts. This fact base was manually constructed and is used to test various scenarios: some authors have no books, all authors published books after their birth, some books have multiple genres, and there are books with both shared and distinct genres.

```
% author(AuthorID, Name, YearOfBirth, CountryOfBirth).
author(1,'John Grisham',1955,'USA').
author(2,'Wilbur Smith',1933,'Zambia').
author(3,'Stephen King',1947,'USA').

% book(Title, AuthorID, YearOfRelease, Pages, Genres).
book('The Firm', 1, 1991, 432,['Legal thriller']).
book('The Runaway Jury',1,1996,414, ['Legal thriller']).
book('The Exchange',1,2023,338,['Legal thriller']).
book('Carrie',3,1974,199,['Horror']).
book('Pet Sematary',3,1983,374,['Horror']).
book('The Shining',3,1977,447,['Gothic novel', 'Horror', 'Psychological
    horror']).
book('Doctor Sleep',3,2013,531,['Horror', 'Gothic', 'Dark fantasy']).
```

Listing 5.26: Sample fact base of authors and books used for testing interconnected data scenarios

One possible way to address this limitation is by integrating *Constraint Logic Programming* (CLP) features provided by SICStus Prolog, specifically through the `clpfd`[6] library. This approach allows the controlled generation of facts by applying domain-specific constraints over variables, ensuring consistency across related data. For example, to generate a consistent author and

[6] Available online at: https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html (last access on June, 2025)

book pair, where the book is only released after the author's birth year, and all attributes follow the format of the fact base, the Listing 5.26 predicate can be defined.

```
generate_author_book(
    author(AuthorID, Name, YearOfBirth, Country),
    book(Title, AuthorID, YearOfRelease, Pages, Genres)
) :-
    arbitrary(string, Title),
    arbitrary(integer, Pages),
    pick_genres(Genres),
    YearOfRelease in 1950..2025,
    MinAge #= 10,
    YearOfRelease #>= YearOfBirth + MinAge,
    labeling([random], [YearOfRelease]).
```

Listing 5.27: Generating book facts based on author ones using CLP

Additionally, genres can be selected dynamically from a predefined list of unique strings, using constraints to ensure variation and uniqueness, as shown in Listing 5.28.

```
:- use_module(library(lists)).

available_genres(['Horror', 'Thriller', 'Fantasy', 'Drama', 'Sci-fi']).

pick_genres(Genres) :-
    domain([N], 1, 3),
    labeling([random], [N]),

    length(Indices, N),
    domain(Indices, 1, 5),
    all_different(Indices),
    labeling([random], Indices),

    available_genres(All),
    pick_genres_from_indices(Indices, All, Genres).

pick_genres_from_indices([], _, []).
pick_genres_from_indices([I|Is], All, [G|Gs]) :-
    nth_genre(I, All, G),
    pick_genres_from_indices(Is, All, Gs).

nth_genre(Index, List, Genre) :-
    nth1(Index, List, Genre).
```

Listing 5.28: Generating unique genres for books generator

These examples show that using CLP to programmatically create structured and semantically sound fact bases is feasible. However, it is not easy to create a complete pipeline that can manage referential consistency, guarantee realistic data distribution, and facilitate reuse across many test situations. The possibility of modularizing and integrating generation into the testing infrastructure to support PBT with associated data is still an open challenge for future research.

5.8 Summary

This chapter demonstrated how fundamental aspects like input generation, shrinking, and composition may be translated onto the declarative paradigm of SICStus Prolog through the design and implementation of a Property-Based Testing library. Custom generators, weighted and dependent generation algorithms, and a simple yet effective shrinking mechanism based on term structure traversal were important elements. In order to demonstrate the expressiveness and compactness that Prolog can provide, the prototype also investigated the reuse of logical relations to construct requirements and test scenarios. The solution presented in this dissertation makes significant enhancements, drawing inspiration from the SWI-Prolog Property-Based Testing library. Complex generator composition, meta-properties, knowledge base manipulation, counterexample reporting, and targeted input generation were not supported in the original. These drawbacks are addressed by the new library, which provides a more extensible and expressive foundation appropriate for educational automation and logic programming.

In response to Research Question RQ1, the best algorithms for instance generation and shrinking in Property-Based Testing in SICStus Prolog are those that use external, recursive shrinking mechanisms in conjunction with independent, isolated fact generation. These techniques preserve user-defined invariants for both basic and compound data types while methodically reducing counterexamples to their simplest failed versions. Because such algorithms are modular, specialized shrinkers that maximize the minimization process can be created, improving the effectiveness and clarity of testing. Regarding the Research Question RQ2, PBT algorithms can benefit from the intrinsic qualities of logic programming and Prolog's non-deterministic nature, especially its capacity to investigate several solutions via backtracking and the adaptability of variable instantiation. This increases the expressiveness and accuracy of tests by allowing the introduction of meta-properties that evaluate the number of potential solutions and validate variable instantiation states. Additionally, the effective integration of data production and property verification, both essential for the successful implementation of PBT in logic programming languages, is facilitated by dynamic manipulation of the fact base and depth-first search.

At the end of the chapter, an extendable testing framework that works with Prolog code was defined. The implementation creates a basis for PBT in Prolog, but further testing is necessary to evaluate its usefulness and efficacy in more complex scenarios. Applying the library to real-world case studies, assessing its advantages and disadvantages in real-world situations, and spotting areas for improvement are all part of the next chapter's approach.

Chapter 6

Application to an Automatic Correction System

This chapter presents a case study on applying Property-Based Testing in real educational programming correction system. It explains how the testing framework has been modified for use with real exam data and goes over the procedure and configuration for assessing student submissions. The outcomes demonstrate the advantages and difficulties of putting this strategy into effect and it provides insights into the real-world implementation of the previously presented features.

6.1 Motivation

In programming courses, it is common to include practical assessments where students are asked to write code. While the correctness of such code can be evaluated statically, it is generally more effective to use automated methods to verify whether it behaves as expected at runtime. Unit tests are the most commonly used approach in these scenarios due to their simplicity and ease of implementation.

At the Faculty of Engineering of the University of Porto (FEUP), the course Functional and Logic Programming (PFL) includes a module on Logic Programming, in which SICStus Prolog is used as the supported interpreter. The course's tests are practical in nature: students receive a set of questions and are required to implement predicates that perform specific tasks and exhibit certain behaviors.

At the moment, the main tool used to support the assessment of each student's answer is unit testing. Test results help direct the evaluation process by showing if the expected behaviors are met, even though grading is not fully automated. However, as discussed in Section 2.1.2, one of the main limitations of unit testing is the effort required to design comprehensive test cases. Despite this effort, it is still easy to overlook edge cases.

As is typical in assessment scenarios, the evaluation process includes both public and private test cases. Public tests are provided in the assignment description to guide the development of student solutions. Private tests, which rely on knowledge bases and inputs not disclosed in advance,

are used as the primary basis for grading. The PFL automated evaluation process is supported by a Python-based framework that operates as a layer over the SICStus Prolog interpreter. The workflow includes the following steps:

1. Extraction of each student's code submission for every question from the Moodle platform;
2. Loading of the private unit tests and the reference solution for each question;
3. Running the reference solution with private unit tests;
4. For each student, the submission is executed against the corresponding unit tests;
5. Recording the pass/fail results for each test, based on the comparison between the reference solution and the student's attempt;
6. Calculation of scores occurs either when the submission passes all the tests or if the student's error is a well-identified error previously recognized by the instructors, with a clear pattern of success/failure;
7. Recording of the test case results in the Moodle platform, alongside the score, if it was possible to automatically compute it.

Figure 6.1 provides an overview of the current correction system workflow in PFL for each student submission.

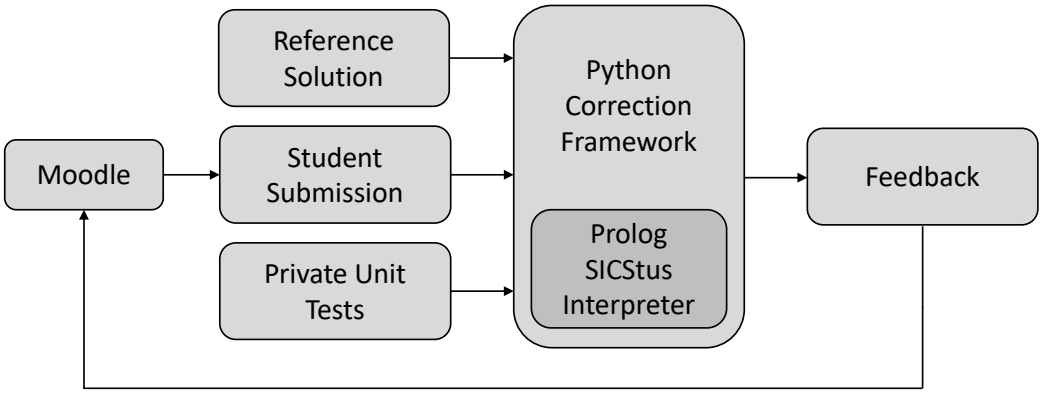

Figure 6.1: Overview of the current evaluation workflow using Unit Testing

Technical restrictions do not, however, preclude the release of these confidential test cases or the evaluation's knowledge bases. Similar to how solutions for individual exercises are frequently exchanged, they can be readily published on the Moodle platform when judged pedagogically beneficial.

The purpose of this case study is to adapt the Python framework to integrate the Property-Based Testing library developed earlier in this dissertation. Properties were written for a sample of

illustrative exercises as a proof-of-concept. The correction system was then re-run using the same set of student submissions. This enabled a direct comparison between unit testing and property-based testing in terms of execution time, detection accuracy, and discrepancies between the two methods. For instance, PBT may be considered superior when it successfully falsifies a property on a student submission that passes all unit tests. Additionally, the study explores how PBT can enhance the feedback provided to students by revealing specific cases where their solutions fail, offering deeper insight into their mistakes.

The creation of properties and, consequently, generators, although potentially generic, still requires a certain degree of problem abstraction and cognitive effort, which can be comparable to writing unit tests. Although PBT presents theoretical advantages, it may not be useful in all scenarios, limiting its adoption in practice. Since each exercise used in the case study has a manually validated solution, the case study was later expanded to Convergence Analysis in Section 6.4.

6.2 Setup

As mentioned in the previous section, the purpose of this case study was to compare both quantitatively and qualitatively the results obtained using the two testing approaches analyzed: Unit Testing and Property-Based Testing. To support this, the Professors provided the exam prompts from the regular and resit periods of the 2024 edition of the PFL course, along with the official solutions for each question and all student submissions, which had been anonymized beforehand. Conducting this case study with real-world data in an authentic context allows for an assessment of the impact that the PBT approach might have had if it had been used in the grading.

Given the impracticality of developing properties, generators, and shrinkers for every exercise from all exams, due to the significant manual effort and time required, a representative sample of exercises was selected to reflect the typical variety found in such assessments. The exercises were chosen based on the following criteria:

- The type of exercise is representative of typical practical programming problems;
- They involve manipulation of various data types and structures;
- They have at least 100 student submissions, ensuring a meaningful sample size;
- They do not fall within the theoretical limitations of the library previously discussed in Section 5.7, namely, they do not involve knowledge bases with complex interdependencies or I/O-based exercises.

In the end, four types of exercises were selected: those involving integers, matrices, databases, and graphs. As such, the Risk R7 mitigation strategy presented in Table 4.1 was applied. This risk refers to the limited availability of evaluation data, since the relevant university course on logic programming is only offered in the first semester, restricting the number of students and test cases. A theoretical analysis confirmed that selecting a representative sample of real-world

exercises is both necessary and sufficient to evaluate the effectiveness of the implemented library in the majority of scenarios found in academic Prolog-based automatic correction systems.

Subsequently, the Python framework was adapted to be able to run the new properties of the library under evaluation. Figure 6.2 presents a high-level overview of the architecture used and the data flow that was required.

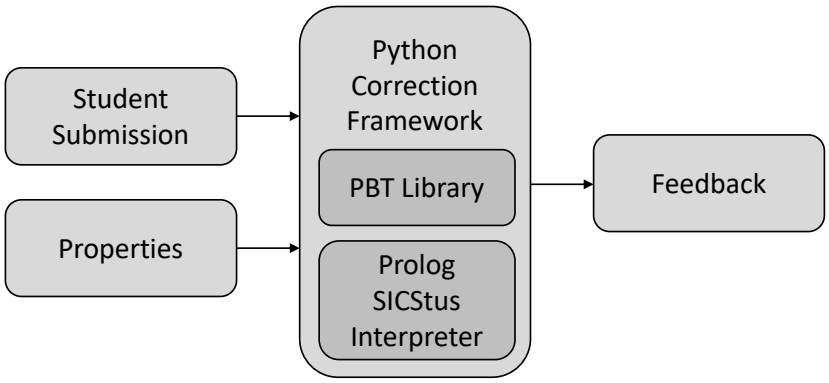

Figure 6.2: Overview of the proposed evaluation workflow using Property-based Testing

For each student and each question, both the unit tests and the property-based tests were executed. The execution times of each test and each property, respectively, were recorded using the Python framework, along with the feedback generated by each testing approach. All PBT tests were performed with shrinking enabled and a number of test cases generation equal to 100, which is the default setting in the library. All measurements were performed on a 2024 MacBook Air M3 (4.05 GHz, 16 GB of RAM), using a single core. The SICStus Prolog interpreter version was 4.9.0, which was the latest available at the time the requirements were gathered, and Python 3.9 was used.

Although execution time is not directly comparable between UT and PBT due to their different natures, it was measured in both cases to provide a general sense of the temporal scale each method requires to evaluate the same submission or entire exam. Likewise, comparison with other existing SICStus Prolog libraries is not feasible, as this library is the first of its kind to offer these capabilities. However, the results may serve as a reference for future comparisons with any subsequent libraries that build upon this work.

6.3 Results and Observations

This section shows the results of applying Property-Based Testing and Unit Testing to the selected exam exercises. In addition to highlighting the relative performance of each method, the study also identifies specific insights and limitations encountered in certain types of problems.

The execution times presented correspond to the average time taken in milliseconds by each individual unit test or by the evaluation of a single input generated through Property-Based Testing,

including the generation and shrinking stages, if applied. These measurements were calculated exclusively for cases where the framework successfully invoked the target predicate: student submissions containing an implementation of the method without syntax errors. Cases resulting in timeouts were excluded from the average timing analysis for both UT and PBT.

Since the analysis is based on real-world data, all available information was reported comprehensively. The total number of exercises, the number of exercises containing errors, those resulting in timeouts, and those executing successfully were recorded. For both timeout and successful cases, the average unit execution time as well as the total testing time were reported separately for each category, as well as combined for the entire dataset. These details are presented in the tables for each exercise.

This approach allowed the identification of performance patterns and informed the decision to limit the number of tests when timeouts are detected, to maintain temporal efficiency without compromising evaluation quality. The recording of errors and timeout occurrences provided valuable insights into common student mistakes and the robustness of the evaluation methodology.

Before presenting the results for each of the selected exercises, it is important to clarify that a direct comparison between UT and PBT is not entirely fair. The effort required to create the tests varied significantly between the two approaches and also depended on the complexity of the exercise. For simpler exercises, the development of unit tests took approximately 5 to 10 minutes, while more complex problems, such as those involving graphs or intricate data structures, required at least 20 minutes to build a battery of tests. All unit tests for all exercises are available in Appendix A. In contrast, the property-based testing approach underwent two to three iterations, each taking over 20 minutes, before reaching the final evaluated version, reflecting a higher degree of maturation, refinement, and cognitive effort.

Although it took longer to create PBT than UT, the iterative procedure resulted in more robust and expressive properties. Therefore, the objective was to determine whether, given a particular amount of effort in test preparation, each strategy was sufficiently expressive to find pertinent problems rather than to directly evaluate the intrinsic performance of each technique in detecting errors. Making this distinction is crucial to accurately interpreting the results.

This analysis also motivates the convergence study in Section 6.4, which investigates whether it is possible to combine the low cognitive load of unit testing with the coverage efficiency of Property-Based Testing.

6.3.1 Integers Processing

The predicate *dec2bin(+Value, -BinaryList, +Size)* converts positive integers into binary lists of a specified size, returning only one solution. If the number is not positive or the size is insufficient, the predicate should fail. An example implementation is shown in Listing 6.1.

```
1 dec2bin_aux(0, []):- !.
2 dec2bin_aux(Dec, [Bit|T]):-
3     Bit is Dec mod 2,
```

```
    Next is Dec div 2,
    dec2bin_aux(Next, T).

dec2bin(Dec, List, N):-
    Dec >= 0,
    dec2bin_aux(Dec, Aux),
    reverse(Aux, Temp),
    length(Temp, Used),
    Needed is N - Used,
    Needed >= 0,
    length(Aux2, Needed),
    maplist( =(0), Aux2),
    append(Aux2, Temp, List).

?- dec2bin(7, List, 8).
List = [0,0,0,0,0,1,1,1] ? ;
no
?- dec2bin(1234, List, 8).
no
```

Listing 6.1: Example implementation of the *dec2bin* predicate used in the evaluation

When creating these tests, the reference implementation provided helpful advice, especially by emphasizing edge cases and crucial characteristics like managing incorrect inputs, correctly padding binary lists, and the output structure for small, large, even, and odd integers. These observations informed the property definitions, ensuring they captured not only correctness but also robustness across a broad input space.

Table 6.1 shows the distribution of submission types for the *dec2bin* exercise, including the number of valid submissions as well as the most common categories of invalid ones, such as syntax errors, missing implementations, and timeouts.

Table 6.1: Distribution of submission types for the *dec2bin* exercise

Parameter	**Count**
Total Submissions Evaluated	329
Invalid Submissions	
– Syntax Errors	62
– Missing Implementations	66
– Timeouts	48
Valid Submissions	153

The Table 6.2 shows that, on average, a unit test is faster than a property-based test, as expected, given the absence of input generation or shrinking in the first testing type.

Table 6.2: Summary of evaluation statistics for valid *dec2bin* submissions

Parameter	**Unit Testing**	**Property-Based Testing**
Number of Tests / Properties	19	11
Number of Valid, Fully Correct Submissions	59	54
Number of Valid Submissions Not Fully Correct	94	99
Single Test Average Execution Time (ms)	51.2	142.4
Total Evaluation Time for All Submissions (s)	65.1	272.5

There were no cases where a unit test failed while all property-based tests passed, but the opposite did occur. In fact, there were five instances, each in a different student submission, where the unit tests fully approved the student's solution, but PBT revealed counterexamples to the implementation. The properties that were successfully falsified in these situations can be found in Listing 6.2.

```
% Cases [1, 1, 1, 1, 1, 1]
prop_dec2bin_max(Size:integer) :-
    MaxValue is 2 ^ Size - 1,
    dec2bin(MaxValue, List, Size),
    maplist(=(1), List).

% Cases [1, 0, 0, 0, 0, 0]
prop_dec2bin_power(Size:integer) :-
    RawValue is 2 ^ (Size - 1),
    Value is max(0, RawValue),
    dec2bin(Value, [1 | Tail], Size),
    maplist(=(0), Tail).
```

Listing 6.2: Key edge case properties for *dec2bin* used in student solution evaluation

It became evident that, although the unit tests covered most scenarios, they did not provide full coverage, particularly for values near the upper bounds or when dealing with powers of two. PBT was able to uncover edge cases in some student submissions, making its test coverage superior for this exercise. Despite a slightly higher average execution time, PBT proved to be the more appropriate approach for evaluating this specific problem.

6.3.2 Matrices

This is an important example, as lists are the fundamental compound type in Prolog for constructing more advanced data structures. Accordingly, the PFL course places strong emphasis on them and evaluates their use in detail across all mini-tests. In this exercise about lists of variable size, content, and dimension, the predicate *matches(+List, +Code)* must succeed if and only if the

configuration represented by *List* matches the given *Code*. Here, *List* is a list of sublists, each representing a dial of the codex. All sublists are expected to have the same length and contain digits representing possible positions of a dial. The second argument, *Code*, is a flat, one-dimensional list of digits, where each digit is expected to match the first element of the corresponding sublist in *List*.

In other words, the predicate succeeds if for every position *i*, the first element of the *i*-th sublist in *List* is equal to the *i*-th element of *Code*. The predicate should fail if:

- *Code* and *List* do not have the same length;
- any digit in *Code* does not match the first element of the corresponding sublist in *List*.

This logic is illustrated in Listing 6.3, where successful matches align the first digits of each sublist in *List* with the elements of *Code*.

```
matches([], []).
matches([[H|_] | Code], [H|Key]):-
        matches(Code, Key).

?- matches([ [3,6,8], [5,0,7], [2,1,4] ], [3,5,2]).
yes
?- matches([ [3,6,8], [5,0,7], [2,1,4] ], [6,0,1]).
no
?- matches([ [3,6,8], [5,0,7], [2,1,4] ], [3,5,2,1]).
no
```

Listing 6.3: Example queries for the *matches* predicate

Given this problem, and considering that both arguments are inputs, the only feasible property for the method is the simple one: given two inputs, the student's submission must produce the same output as the reference solution used in Unit Testing as well. However, it is also necessary to take into account the need to generate valid inputs, inputs that are invalid due to list size, and inputs that are invalid due to list content.

Therefore, three different generators were created, one for each case, and a single property that randomly chooses one of the generators to ensure full coverage. For simplicity in constructing the property, and because in this case a generator cannot be directly associated with a higher likelihood of catching errors or achieving better coverage, it was assumed that each of the three input generators would be used with equal probability. Assuming *pfl_matches/2* is the reference solution, the property used to evaluate students' submissions is shown in Listing 6.4.

```
prop_matches([ListA | CodeA]:valid_matches,
             [ListB | CodeB]:invalid_matches_size,
             [ListC | CodeC]:invalid_matches_content) :-

```

```
    random_member([List, Code], [
        [ListA | CodeA],
        [ListB | CodeB],
        [ListC | CodeC]
    ]),

    ( call(matches(List, Code)) -> S1 = true ; S1 = false ),
    ( call(pfl_matches(List, Code)) -> S2 = true ; S2 = false ),

    S1 = S2.
```

Listing 6.4: Property used to validate student submissions

Table 6.3 summarizes the submission types recorded for this exercise.

Table 6.3: Distribution of submission types for the *matches* exercise

Parameter	**Count**
Total Submissions Evaluated	111
Invalid Submissions	
– Syntax Errors	6
– Missing Implementations	16
– Timeouts	2
Valid Submissions	87

The results for the remaining 87 valid submissions are presented in Table 6.4, which includes only those that were successfully executed without errors or timeouts.

Table 6.4: Summary of evaluation statistics for valid *matches* submissions

Parameter	**Unit Testing**	**Property-Based Testing**
Number of Tests / Properties	9	1
Number of Valid, Fully Correct Submissions	9	9
Number of Valid Submissions Not Fully Correct	78	78
Single Test Average Execution Time (ms)	73.2	183.4
Total Evaluation Time for All Submissions (s)	13.9	87.5

Unlike the previous exercise, both Unit Testing and Property-Based Testing achieved the same accuracy rate in this case. One important point to highlight is that, despite the clearly higher effort required to build the PBT generators for this scenario, a single property was sufficient to match the effectiveness of the traditional unit tests used.

The execution time, as expected, was higher for PBT and even higher than in the previous exercise. This can be explained by the fact that the generators are more complex and include more internal checks related to size and component validation.

Despite having a higher execution time, it is important to highlight that PBT had similar precision and, in theory, achieved broader scenario coverage.

6.3.3 Knowledge Base Processing

The predicate *next_gen(+Previous, -Next)* receives a list of binary values and computes the next generation by applying the existing rules, *rule/2* facts in the database, to each position in the list. To do this, a sliding window of three bits is considered: the previous bit, the current bit, and the next bit. At the edges of the list, where there are no left or right neighbors, the missing values are assumed to be 0. The next generation is built based on these rule applications, one position at a time, and one example is given in Listing 6.5.

```
update_rule(N):-
        dec2bin(N, Bits, 8),
        abolish(rule/2),
        between(0,1,FirstBit),
        between(0,1,SecondBit),
        between(0,1,ThirdBit),
        Index is 8 - (FirstBit * 4 + SecondBit * 2 + ThirdBit),
        nth1(Index, Bits, Bit),
        assert(rule(FirstBit-SecondBit-ThirdBit, Bit)),
        fail.
update_rule(_).

next_gen(Gen0, Gen1):-
        apply_rules(0, Gen0, Gen1).

apply_rules(Left, [Self], [New]):-
        rule(Left-Self-0, New).
apply_rules(Left, [Self, Right | Rest], [New | Tail]):-
        rule(Left-Self-Right, New),
        apply_rules(Self, [Right|Rest], Tail).

?- update_rule(90).
yes
?- listing(rule/2).
rule(0-0-0, 0).
rule(0-0-1, 1).
rule(0-1-0, 0).
rule(0-1-1, 1).
rule(1-0-0, 0).
```

```
rule(1-0-1, 1).
rule(1-1-0, 0).
rule(1-1-1, 1).
?- next_gen([0,0,1,0,0,0,1,0], NextGen).
NextGen = [1,0,1,0,1,0,1,0]
```

Listing 6.5: Example usage of *next_gen* after updating rules

This predicate interacts with the dynamic database in a simplified manner. To test student submissions, the *update_rule/1* predicate was used as a helper to control the rule set. Four property-based tests were defined, shown in Listing 6.6, after building custom input generators: a generic one with a random binary list and a random rule, one testing only the output size, one testing the minimum rule (0), and another testing the maximum rule (255).

```
% Generic case
prop_next_gen_generic_case(Binary:binary_list, Rule:short_int) :-
    update_rule(Rule),
    pfl_next_gen(Binary, Result),
    next_gen(Binary, Result).

% Size
prop_next_gen_size(Binary:binary_list) :-
    next_gen(Binary, Result),
    length(Binary, N),
    length(Result, N).

% The result is always 0
prop_next_gen_rule_0(Binary:binary_list) :-
    update_rule(0),
    next_gen(Binary, Result),
    maplist(=(0), Result).

% The result is always 1
prop_next_gen_rule_1(Binary:binary_list) :-
    update_rule(255),
    next_gen(Binary, Result),
    maplist(=(1), Result).
```

Listing 6.6: Property-based tests used to validate *next_gen* implementations

Table 6.5 shows the distribution of submission types for the *next_gen* exercise, including the number of valid submissions as well as the most common categories of invalid ones, such as syntax errors, missing implementations, and timeouts.

The results in Table 6.6 once again showed that Unit Testing and Property-Based Testing were equivalent in terms of effectiveness at detecting errors and edge cases in the students' submissions.

Table 6.5: Distribution of submission types for the *next_gen* exercise

Parameter	**Count**
Total Submissions Evaluated	329
Invalid Submissions	
– Syntax Errors	43
– Missing Implementations	102
– Timeouts	23
Valid Submissions	161

Table 6.6: Summary of evaluation statistics for valid *next_gen* submissions

Parameter	**Unit Testing**	**Property-Based Testing**
Number of Tests / Properties	6	4
Number of Valid, Fully Correct Submissions	26	26
Number of Valid Submissions Not Fully Correct	135	135
Single Test Average Execution Time (ms)	157.9	249.1
Total Evaluation Time for All Submissions (s)	50.3	1281.2

In general, fewer property-based tests are needed to achieve the same level of coverage as unit tests, although, as acknowledged at the beginning of this section, property-based tests do require greater cognitive effort, as well as more abstraction and implementation time.

In fact, due to the explicit use of assert and retract operations in the implementation of *update_rule/1*, used in both the unit tests and the property-based tests, which modifies the dynamic knowledge base, there was an increase in the average execution time for both testing strategies.

6.3.4 Graphs

The predicate *largest_circle/1* identifies the largest group of people forming a closed gift exchange circle within the book club. Starting with any person, the algorithm must follow the *gives_gift_to/3* edges until the circle closes. That is, when the last person gives a gift back to the first person in the chain. If multiple groups have the same maximum size, the predicate should yield multiple solutions, one for each group.

In addition to requiring graph traversal, this exercise queries edges stored as facts in the knowledge base, adding an extra layer of complexity for Property-Based Testing, especially when creating suitable generators. An example of a successful usage of the predicate is shown in Listing 6.7.

```
1 gives_gift_to(one, 'New 1', two).
2 gives_gift_to(two, 'New 2', three).
```

```
gives_gift_to(three, 'New 3', four).
gives_gift_to(four, 'New 4', five).
gives_gift_to(five, 'New 5', six).
gives_gift_to(six, 'New 6', seven).
gives_gift_to(seven, 'New 7', one).
gives_gift_to(eight, 'New 8', nine).
gives_gift_to(nine, 'New 9', eight).

?- largest_circle(Group)
Group = [one, two, three, four, five, six, seven] ?
no.
```

Listing 6.7: Example of a valid gift circle with largest size

The library already includes two key generators for this task: a set generator (for unique vertices) and a partition generator (to divide them into potential cycles). The solution used is shown in Listing 6.8.

```
get_set(Size, L) :-
    get_set(Size, [], L).

get_set(Size, Acc, Acc) :-
    length(Acc, Size), !.

get_set(Size, Acc, L) :-
    random_between(1, 1000000, Num),
    ( member(Num, Acc) ->
        get_set(Size, Acc, L)
    ;
        get_set(Size, [Num|Acc], L)
    ).

random_partition(0, []) :- !.
random_partition(Total, [X | Rest]) :-
    Total > 1,
    random_between(1, Total, X), !,
    RestTotal is Total - X,
    random_partition(RestTotal, Rest).

composite(partition, [Total:short_int], Partition) :-
    random_partition(Total, Partition).
```

Listing 6.8: Composite generator using sets and partitions

Given that both generators, the sets and partitions, already include edge cases like large cycles, empty graphs, or disconnected nodes, a single property was sufficient to cover multiple scenarios effectively.

However, the *largest_circle/1* predicate may return more than one solution, as stated in the exercise description, or it may fail altogether if no valid cycles or facts are present. For this reason, although the property remains generic, it needed to account for two possible branches of execution in order to avoid creating separate generators that either always produce valid circles or always trigger errors. The property includes one branch that asserts both predicates fail when no solution is possible, and another that extracts all valid solutions from both the reference and student predicates. These solutions are then sorted to ensure a fair comparison that does not depend on the internal implementation details of the student's code.

It is also necessary to clean the knowledge base before executing the next input generation or invoking the next property, ensuring isolation between test results. This is done using the library's layer responsible for managing dynamic facts, as detailed in Section 5.5.

As shown in Listing 6.9, this results in a more complex property, but one capable of achieving comprehensive coverage.

```
prop_large_circle_generic_case(Partition:partition) :-
    sum(Partition, Total),
    get_set(Total, Set),
    partition_by_sizes(Set, Partition, GiftsPartition),
    set_gift_graph(GiftsPartition, _Graph),

    % Try to collect all solutions
    ( findall(G, pfl_largest_circle(G), ReferenceGroups), StudentGroups
        \== [] ->
        findall(GR, largest_circle(GR), StudentGroups),
        sort(ReferenceGroups, SortedRef),
        sort(StudentGroups, SortedStudent),
        SortedStudent == SortedRef
    ;
        % Both predicates are expected to fail
        \+ largest_circle(_),
        \+ pfl_largest_circle(_)
    ),

    clear_graph.
```

Listing 6.9: Generic property for validating *largest_circle*

Table 6.7 shows the distribution of submission types for the *largest_circle* exercise, including the number of valid submissions as well as the most common categories of invalid ones, such as syntax errors, missing implementations, and timeouts.

Table 6.7: Distribution of submission types for the *largest_circle* exercise

Parameter	**Count**
Total Submissions Evaluated	329
Invalid Submissions	
– Syntax Errors	22
– Missing Implementations	129
– Timeouts	52
Valid Submissions	126

Table 6.8 includes the evaluation summary only for those submissions that were successfully executed without errors or timeouts.

Table 6.8: Summary of evaluation statistics for valid *largest_circle* submissions

Parameter	**Unit Testing**	**Property-Based Testing**
Number of Tests / Properties	9	1
Number of Valid, Fully Correct Submissions	22	22
Number of Valid Submissions Not Fully Correct	104	104
Single Test Average Execution Time (ms)	132.7	530.4
Total Evaluation Time for All Submissions (s)	43.4	732.2

Both Unit Testing and Property-Based Testing were equally effective in validating student submissions. However, PBT had a significantly higher execution time due to the overhead of storing and querying dynamically generated graphs. Despite equal validation capabilities, it's important to note that writing unit tests manually for large cycles, missing cycles, or dense graphs would be impractical. Hence, PBT proves more scalable and practical for this and similar scenarios.

6.4 Convergence Analysis

Although PBT offers an expressive approach to program verification, its practical application often requires a high degree of abstraction, particularly in the design of properties and the construction of generators. This cognitive effort can be comparable to that of writing unit tests, potentially limiting the widespread adoption of PBT in educational or time-constrained contexts. To explore this trade-off, the case study was expanded with a convergence analysis based on the hypothesis that a simple property, one that compares the student's output directly to that of the reference solution, could offer meaningful insights. By systematically varying the number of input generations (N), and using random strategies, the goal was to determine whether there exists a threshold for N beyond which the verdicts of Property-Based Testing and Unit Testing align. Identifying such a threshold

would allow instructors to benefit from the automation and coverage of PBT without incurring excessive design overhead.

To conduct this convergence experiment, a random generation strategy was adopted to create diverse inputs, allowing the basic property to serve as an intuitive and robust baseline aligned with the intended behavior. The four previous exercises were used in the experiment along with their corresponding trivial properties. It is worth noting that in some cases, the properties used in the earlier sections were reused, as they were already trivial in nature. Listing 6.10 shows the trivial properties used in the study.

```
% dec2bin
prop_dec2bin_generic_case(Size:integer) :-
    get_random_number(Size, Decimal),
    dec2bin(Decimal, List, Size),
    pfl_dec2bin(Decimal, List, Size).

% matches
prop_matches_generic_case([ListA | CodeA]:valid_matches,
             [ListB | CodeB]:invalid_matches_size,
             [ListC | CodeC]:invalid_matches_content) :-

    random_member([List, Code], [
        [ListA | CodeA],
        [ListB | CodeB],
        [ListC | CodeC]
    ]),
    ( call(matches(List, Code)) -> S1 = true ; S1 = false ),
    ( call(pfl_matches(List, Code)) -> S2 = true ; S2 = false ),
    S1 = S2.

% next_gen
prop_next_gen_generic_case(Binary:binary_list, Rule:short_int) :-
    update_rule(Rule),
    pfl_next_gen(Binary, Result),
    next_gen(Binary, Result).

% largest_circle
prop_large_circle_generic_case(Partition:partition) :-
    sum(Partition, Total),
    get_set(Total, Set),
    partition_by_sizes(Set, Partition, GiftsPartition),
    set_gift_graph(GiftsPartition, _Graph),

    ( findall(G, pfl_largest_circle(G), ReferenceGroups), StudentGroups
       \== [] ->
```

```
35          findall(GR, largest_circle(GR), StudentGroups),
36          sort(ReferenceGroups, SortedRef),
37          sort(StudentGroups, SortedStudent),
38          SortedStudent == SortedRef
39      ;
40          % Both predicates are expected to fail
41          \+ largest_circle(_),
42          \+ pfl_largest_circle(_)
43      ),
44
45      clear_graph.
```

Listing 6.10: Trivial properties comparing student and reference implementation

This part of the case study was conducted only with the valid submissions for the exercise in question, that is, those in which the predicate to be tested was implemented by the student and no syntax errors or timeouts were detected by the framework. This ensured that the resulting values of N influenced only the likelihood of falsifying the property.

For each valid submission and for each exercise, the number of input generations N applied for the trivial property was incrementally increased. For valid and fully correct submissions, the value of N is irrelevant when it comes to determining when the effectiveness of PBT with a simple property matches that of Unit Testing, as the property would never be falsified. On the other hand, for the remaining submissions, the value of N at which the property was first falsified was recorded. Table 6.9 shows the minimum, maximum, and average values of N required to falsify the trivial property in each exercise.

Table 6.9: Summary statistics for metric N across the exercises

Exercise	**Min. N**	**Max. N**	**Average N**
dec2bin	4	37	10.3
matches	1	9	2.7
next_gen	2	6	3.4
largest_circle	1	5	2.2

The data show that the more complex the exercise or the data structure it manipulates, the faster PBT can match UT in terms of bug detection, and as demonstrated in previous sections, it can even outperform UT. While the first exercise deals with integers, covering a wide range and thus making it less likely for the generator to quickly falsify the property, the graph exercise involves a more rigid structure, which makes it possible to falsify the property with fewer sequential runs.

Although the data do not provide a strong basis for generalization, they suggest that more complex exercises, those require more effort to define generators and properties, tend to require lower values of N to achieve adequate coverage using the trivial property. As such, the tester may

eventually reduce the value of N while still ensuring sufficient test coverage, leading to a shorter overall execution time and partially offsetting the time invested in generator creation.

These values strongly depend on the specific exercise and on how much time is available to assess and correct student submissions, indicating that there is no universally applicable value for N. It is likely that the required N could be drastically reduced if, instead of random input generation, a targeted input generation approach was used. This method would leverage specific preconditions such as edge cases, extreme values, very small, very large, zeros, and powers of two, as discussed in Section 6.3.1.

Overall, the strategy based on simple random input generation has important limitations. First, there is a chance of failing to falsify the property that should be detected due to the probabilistic nature of the method, although this can be mitigated by targeted input generation. Additionally, the trivial property is binary, tests either pass or fail, which may not capture the necessary granularity to evaluate varying quality levels of student solutions. This also implies risks of false negatives, as the evaluation relies on probabilistic sampling. In this context, a false negative is considered to occur when the property for a predicate containing a bug is not falsified by PBT because, due to its random nature, the input generation happened to produce a case that did not trigger the failure.

While using the trivial property with varying input counts offers valuable insights into the convergence between Unit Testing and Property-Based Testing, its practical application requires caution and ideally should be combined with more refined properties to ensure robustness and accuracy in assessment.

6.5 Summary

In this chapter, Property-Based Testing was presented as a complement to Unit Testing, offering notable advantages, particularly in educational contexts involving multiple submissions. Its ability to automatically generate diverse and unexpected inputs enables the detection of failures that conventional unit tests often overlook, especially in edge or boundary cases. Furthermore, PBT can reduce maintenance effort, as well-defined properties tend to generalize more effectively than fixed test sets.

One important conclusion is that it takes time, maturity, and improvement to achieve adequate test coverage, whether using UT or PBT. Furthermore, given that unit testing can accomplish comparable coverage with less time and cognitive effort, it may be more appropriate for specific exercise kinds, especially simpler ones. On the other hand, despite the greater initial effort, property-based testing might offer superior coverage for increasingly complicated issues involving sophisticated data structures.

Nevertheless, PBT introduces challenges. Crafting meaningful properties and appropriate generators requires a higher level of abstraction and greater initial cognitive effort, which can hinder adoption in environments with limited time or resources. In more complex tasks, such as those involving graphs or dynamic structures, performance may also be impacted due to the computational cost of generating and validating intricate scenarios. Convergence analysis indicates

that, even with simple properties, PBT can achieve alignment with UT after a sufficient number of test generations. This suggests a promising hybrid approach, where a moderate application of property-based tests can offer robust automatic validation, especially when guided generation strategies are employed. Such a balance between setup effort and evaluation quality makes PBT particularly suitable for automated feedback systems in programming education.

The next chapter consolidates these findings into broader conclusions regarding the role of automated verification in supporting the teaching of logic programming.

Chapter 7

Conclusions

This chapter presents the conclusions of the work developed in this dissertation, the objectives achieved, the main contributions of the research, and the future work that can be addressed in subsequent iterations.

7.1 Summary of Findings

The integration of Property-Based Testing into the Prolog programming language was examined in this dissertation, laying the groundwork for its use in automated evaluation and educational settings. The first chapters outlined the theoretical framework, looking at how important ideas in programming language paradigms and software testing affect the development and suitability of PBT methodologies. The logical paradigm and non-deterministic execution model of Prolog, which present both special opportunities and difficulties when modifying PBT approaches customarily created for functional or imperative languages, made it stand out in this analysis.

In response to RQ1, the best algorithms for instance generation and shrinking in Property-Based Testing within SICStus Prolog are those that use external, recursive shrinking mechanisms combined with independent and isolated facts and input generation. These techniques preserve user-defined invariants for both basic and compound data types while systematically minimizing counterexamples to their simplest failing forms. Their modular nature allows the creation of specialized shrinkers that enhance the minimization process, improving the effectiveness and clarity of test feedback.

Regarding RQ2, the selected algorithms benefit from the intrinsic qualities of logic programming and Prolog's non-deterministic nature, especially its capacity to explore multiple solutions through backtracking and the flexible instantiation of variables. This increases the expressiveness and accuracy of tests by enabling meta-properties that assess the number of possible solutions and validate variable instantiation states. The dynamic manipulation of the fact base, together with depth-first search, facilitates the seamless integration of data generation and property verification, both essential for successful Property-Based Testing in logic programming languages.

In response to RQ3, Property-Based Testing is shown to be a valuable complement to Unit Testing, particularly in educational contexts involving multiple submissions. Its ability to automatically generate diverse and unexpected inputs helps detect failures that conventional unit tests often miss, especially in edge or boundary cases. Furthermore, Property-Based Testing can reduce maintenance effort, since well-defined properties tend to generalize more effectively than fixed test sets. However, there are drawbacks in using this testing approach. Crafting meaningful properties and suitable generators demands a higher level of abstraction and initial cognitive effort, which may hinder adoption in environments with limited time or resources. The computational cost of creating and testing complex scenarios can have an impact on performance for more complex properties, including those involving graphs or dynamic data structures. The convergence analysis indicates that even simple properties can align Property-Based Testing results with Unit Testing outcomes after sufficient test case generations. This points to a possible hybrid strategy that can offer reliable automatic validation through the moderate use of property-based tests, particularly when directed generation strategies are used. Property-Based Testing is especially well-suited for automated feedback systems in programming education because it strikes a compromise between setup time and evaluation quality.

Despite drawing inspiration from the Property-Based Testing library for SWI-Prolog that already exists, the suggested solution adds a number of important features that greatly increase its functionality. The initial SWI-Prolog library included a small collection of basic generators with little composition support and no tools for creating intricate, weighted, or dependent input structures. Although shrinking was accessible in a basic form, its applicability in debugging and educational contexts was limited because it did not offer reporting or traceability of counterexamples. Furthermore, a feature that is very crucial in logic programming, dynamic knowledge base manipulation during test execution, was not supported by the original library. Additionally, it did not enable meta-properties like variable instantiation states or limits on the number of valid solutions, and it did not provide a way to generate targeted inputs to direct the testing process toward edge cases. The library created in this dissertation, on the other hand, was specifically created to overcome these constraints, making it more extendable, expressive, and appropriate for situations involving instructional automation.

Thus, this dissertation condenses knowledge about current Property-Based Testing techniques and frameworks and validates the feasibility and advantages of applying them in Prolog through a practical implementation in SICStus Prolog, with a focus on educational applications.

7.2 Main Contributions

This dissertation presents both theoretical and practical contributions to the field of software testing in the context of logic programming, particularly through the adaptation and implementation of Property-Based Testing techniques in Prolog.

From a theoretical perspective, the work provides a comprehensive overview of existing PBT tools and libraries, analyzing them across multiple dimensions, such as the underlying programming paradigms, type systems, and execution models, interpreted or compiled. This classification establishes a broader understanding of the contexts in which PBT is applied and highlights how these characteristics influence the design and effectiveness of test generation and shrinking strategies. Furthermore, a thorough analysis of the most recent input generation and shrinking algorithms was carried out, revealing important design trends, constraints, and how well they matched the characteristics of various language families. Because of its logic-based paradigm and nondeterministic execution approach, this analysis helped identify the algorithms that would work best in Prolog. A discussion of the adaptation of ideas like shrinkage techniques, entropy control, and narrowing in the particular setting of logic programming is another theoretical contribution.

From a practical and applied perspective, the dissertation introduces a novel PBT framework implemented in SICStus Prolog. Custom test case creation, counterexample shrinking, and the defining of attributes appropriate for automated validation are all supported by this framework. In order to maintain user-defined invariants while utilizing Prolog's advantages in backtracking and flexible variable instantiation, it integrates recursive external shrinkage and independent fact generation techniques. In order to compare the framework with conventional Unit Testing methods, it was assessed inside the framework of automatic code grading systems. The findings demonstrate PBT's capacity to identify minute mistakes, particularly in increasingly difficult activities, and to deliver insightful feedback with less maintenance overhead over time. Additionally, the paper proposes a hybrid strategy in which PBT is used in conjunction with UT, especially in educational settings where input scalability and variability are beneficial.

Lastly, this work establishes the groundwork for future expansions, such as the ability to simulate I/O streams, integrate with Constraint Logic Programming to produce semantically consistent data, and support for testing input/output predicates. The goal of these directions is to make the framework more expressive and applicable to increasingly complex educational use cases as well as real-world Prolog programs.

7.3 Future Work

During the elaboration of this dissertation, several points for improvement were identified, both regarding the selection of suitable Property-Based Testing algorithms for Prolog, given its current feature set, and concerning the expansion of the framework to support more complex testing needs.

According to Lo et al. [Lo et al., 2019, Lo et al., 2020], in the context of the Coq theorem prover and typed programming language using the QuickChick framework, genetic algorithms for External Shrinking have proven more effective than random sampling. Given that pure Targeted Input Generation has proven unfeasible for Prolog, as explained in Section 3.2.2, an alternative is to iterate and optimize random input generation techniques to seek improved efficiency and effectiveness in testing, since random generation is more commonly used and easier to implement in this context. In terms of shrinking, developing a SICStus Prolog PRNG library that allows seed

inspection and reproduction, similar to the Hypothesis Python framework, would enable internal shrinking, which is currently blocked by existing tool limitations.

The concept of Narrowing [Antoy et al., 2000], discussed in Section 3.2.1, could play a significant role in Prolog by helping to handle potentially infinite data structures, a capability ensured by libraries like Falsify in their Internal Shrinking process. However, as noted by Cheong and Fribourg [Cheong and Fribourg, 1998], implementing Narrowing in Prolog may present considerable challenges in implementation.

The potential for parallelism in both input generation and shrinking is an interesting avenue to explore. Nevertheless, this possibility may impose restrictions on the choice of algorithms, since some methods do not support parallel execution [Krook et al., 2023]. Additionally, it is important to consider that not all Prolog interpreters support parallelism. SICStus Prolog [Carlsson and Mildner, 2010] does not natively provide parallel execution capabilities, which could limit the implementation of parallel approaches in this environment. In addition, the implementation of a PBT framework in Prolog could directly benefit from incorporating constraint programming techniques [Blanco et al., 2019], potentially improving the efficiency and expressiveness of test data generation and shrinking.

It is important to improve support for predicates involving input/output interactions. Predicates that read from or write to the console cannot be automatically tested by the present framework. Instead, I/O streams must be simulated manually. In order to reduce verbosity and complexity for testers, future work should concentrate on creating a modular and integrated method for simulating and smoothly intercepting user input and output streams.

Improving the generation and management of complex, interconnected knowledge bases is a promising avenue. While individual facts can be generated, maintaining semantic consistency and referential integrity across related data remains challenging. Leveraging Constraint Logic Programming techniques can enable controlled, constraint-driven generation of related facts, ensuring realistic and coherent datasets for academic fields. Developing reusable, modular pipelines that integrate CLP into the testing framework will allow robust Property-Based Testing over complex knowledge bases.

In order to better use Prolog's capabilities and increase the efficacy of Property-Based Testing in real-world applications, a future version of this work that aims to improve and expand the original framework is justified, given the challenges encountered and the opportunities identified.

Bibliography

[Aichernig et al., 2017] Aichernig, B. K., Marcovic, S., and Schumi, R. (2017). Property-based testing with external test-case generators. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation Workshops, 2017*, page 337 – 346. DOI: 10.1109/ICSTW.2017.62.

[Alves et al., 2023] Alves, J., Neto, A. C., Pereira, M. J. V., and Henriques, P. R. (2023). Characterization and identification of programming languages. *12th Symposium on Languages, Applications, and Technologies*, pages 12–13. DOI: 10.4230/OASIcs.SLATE.2023.13.

[Amaral et al., 2014] Amaral, C., Florido, M., and Costa, V. S. (2014). Prologcheck - property-based testing in prolog. In *International Symposium on Functional and Logic Programming, 2014*, volume 8475 LNCS, page 1 – 17. DOI: 10.1007/978-3-319-07151-0_1.

[Antoy et al., 2000] Antoy, S., Echahed, R., and Hanus, M. (2000). A needed narrowing strategy. *Journal of ACM*, 47(4):776–822. DOI: 10.1145/347476.347484.

[Apt, 1996] Apt, K. R. (1996). *From logic programming to Prolog*. Prentice-Hall, Inc., USA. ISBN: 978-0-13-230368-2.

[Bartonícek, 2014] Bartonícek, J. (2014). Programming language paradigms and the main principles of object-oriented programming. *Bulletin of the Centre for Research and Interdisciplinary Study*, 12. DOI: 10.2478/cris-2014-0006.

[Benac Earle et al., 2016] Benac Earle, C., Fredlund, L.-r., and Hughes, J. (2016). Automatic grading of programming exercises using property-based testing. In *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '16, page 47–52, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/2899415.2899443.

[Bereczky et al., 2020] Bereczky, P., Horpácsi, D., Köszegi, J., Szeier, S., and Thompson, S. (2020). Validating formal semantics by property-based cross-testing. In *Proceedings of the ACM International Conference Proceeding Series*, page 150 – 161. DOI: 10.1145/3462172.3462200.

[Bhattacharjee et al., 2018] Bhattacharjee, K., Maity, K., and Das, S. (2018). A search for good pseudo-random number generators : Survey and empirical studies. *Computing Research Repository*. DOI: 10.48550/arXiv.1811.04035.

[Bissi et al., 2016] Bissi, W., Neto, A. G. S. S., and Emer, M. C. F. P. (2016). The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, 74:45–54. DOI: 10.1016/j.infsof.2016.02.004.

[Blanco et al., 2019] Blanco, R., Miller, D., and Momigliano, A. (2019). Property-based testing via proof reconstruction. In *ACM International Conference Proceeding Series*. DOI: 10.1145/3354166.3354170.

[Cardelli and Wegner, 1985] Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–523. DOI: 10.1145/6041.6042.

[Carlsson and Mildner, 2010] Carlsson, M. and Mildner, P. (2010). Sicstus prolog - the first 25 years. *Theory and Practice of Logic Programming*, 12. DOI: 10.1017/S1471068411000482.

[Chalup and Maire, 1999] Chalup, S. and Maire, F. (1999). A study on hill climbing algorithms for neural network training. In *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*, volume 3, pages 2014–2021 Vol. 3. DOI: 10.1109/CEC.1999.785522.

[Chen et al., 2017] Chen, Z., O'Connor, L., Keller, G., Klein, G., and Heiser, G. (2017). The cogent case for property-based testing. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, pages 1–7. ACM. DOI: 10.1145/3144555.3144556.

[Chen et al., 2022] Chen, Z., Rizkallah, C., O'Connor, L., Susarla, P., Klein, G., Heiser, G., and Keller, G. (2022). Property-based testing: Climbing the stairway to verification. In *Proceedings of the 15th ACM International Conference on Software Language Engineering, 2022*, page 84 – 97. DOI: 10.1145/3567512.3567520.

[Cheong and Fribourg, 1998] Cheong, P. and Fribourg, L. (1998). Implementation of narrowing: The prolog-based approach. *Logic Programming Languages: Constraints, Functions, and Objects*. DOI: 10.7551/mitpress/4313.003.0004.

[Claessen and Hughes, 2000] Claessen, K. and Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the 5th ACM International Conference on Functional Programming, 2000*, pages 268–279. ACM. DOI: 10.1145/351240.351266.

[Cohen, 1985] Cohen, J. (1985). Describing prolog by its interpretation and compilation. *Communications of the ACM*, 28(12). DOI: 10.1145/214956.214960.

[Corgozinho et al., 2023] Corgozinho, A. L., Valente, M. T., and Rocha, H. (2023). How developers implement property-based tests. In *IEEE International Conference on Software Maintenance and Evolution, 2023*, page 380 – 384. DOI: 10.1109/ICSME58846.2023.00049.

[Costa et al., 2011] Costa, V., Damas, L., and Rocha, R. (2011). The yap prolog system. *Computing Research Repository*, 12. DOI: 10.1017/S1471068411000512.

[De Angelis et al., 2019] De Angelis, E., Fioravanti, F., Palacios, A., Pettorossi, A., and Proietti, M. (2019). Property-based test case generators for free. In *Tests and Proofs: 13th International Conference, TAP 2019, Held as Part of the Third World Congress on Formal Methods 2019, Porto, Portugal, October 9–11, 2019, Proceedings*, page 186–206, Berlin, Heidelberg. Springer-Verlag. DOI: 10.1007/978-3-030-31157-5_12.

[de Vries, 2023] de Vries, E. (2023). falsify: Internal shrinking reimagined for haskell. In *16th ACM SIGPLAN International Haskell Symposium, 2023*, Haskell 2023, page 97–109, New York, NY, USA. ACM. DOI: 10.1145/3609026.3609733.

[Duran and Ntafos, 1984] Duran, J. W. and Ntafos, S. C. (1984). An evaluation of random testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444. DOI: 10.1109/TSE.1984.5010257.

[Fink and Bishop, 1997] Fink, G. and Bishop, M. (1997). Property-based testing: a new approach to testing for assurance. *Special Interest Group on Software Engineering Notes*, 22:74–80. DOI: 10.1145/263244.263267.

[Floyd, 1979] Floyd, R. W. (1979). The paradigms of programming. *Communications of the ACM*, 22:455–460. DOI: 10.1145/359138.359140.

[Geruslu et al., 2020] Geruslu, V., Rainer, A., Lauvas, P., and Arcuri, A. (2020). Software-testing education: A systematic literature mapping. *Journal of Systems and Software*, 165:110570. DOI: 10.1016/J.JSS.2020.110570.

[Gissurarson et al., 2022] Gissurarson, M. P., Applis, L., Panichella, A., van Deursen, A., and Sands, D. (2022). Propr: Property-based automatic program repair. In *IEEE/ACM 44th International Conference on Software Engineering, 2022*, pages 1768–1780. DOI: 10.1145/3510003.3510620.

[Goldstein et al., 2024] Goldstein, H., Cutler, J. W., Dickstein, D., Pierce, B. C., and Head, A. (2024). Property-based testing in practice. In *Proceedings - International Conference on Software Engineering*, page 2307 – 2319. DOI: 10.1145/3597503.3639581.

[Goldstein et al., 2023a] Goldstein, H., Frohlich, S., Wang, M., and Pierce, B. C. (2023a). Reflecting on random generation. *Proceedings of the ACM on Programming Languages*, 7:28 – 31. DOI: 10.1145/3607842.

[Goldstein and Pierce, 2022] Goldstein, H. and Pierce, B. C. (2022). Parsing randomness. *Proceedings of the 15th ACM International Conference on Software Language Engineering, 2022*, 6(OOPSLA2). DOI: 10.1145/3563291.

[Goldstein et al., 2023b] Goldstein, H., Pierce, B. C., and Head, A. (2023b). Tyche: In situ analysis of random testing effectiveness. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. DOI: 10.1145/3586182.3615788.

[Graeff et al., 2024] Graeff, G. R., de Mello, B. A., Duarte, D., de Almeida Sampaio Braga, A., and da Silva Feitosa, S. (2024). Random generation of haskell programs applied to optimization testing in compilers; [geração aleatória de programas haskell aplicada a testes de otimizações em compiladores]. *Revista de Informatica Teorica e Aplicada*, 31:20 – 29. DOI: 10.22456/2175-2745.134925.

[Hamlet and Taylor, 1990] Hamlet, D. and Taylor, R. (1990). Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411. DOI: 10.1109/32.62448.

[Hritcu et al., 2013] Hritcu, C., Lampropoulos, L., Spector-Zabusky, A., de Amorim, A. A., Maxime Dénès, J. H., Pierce, B. C., and Vytiniotis, D. (2013). Testing noninterference, quickly. In *Proceedings of the 18th ACM International Conference on Functional Programming*, pages 455–468. ACM. DOI: 10.1145/2500365.2500574.

[Hughes, 2011] Hughes, J. (2011). Specification based testing with quickcheck. In *2011 Formal Methods in Computer-Aided Design*, pages 17 – 39. DOI: 10.1007/978-1-349-91518-7_2.

[James, 1990] James, F. (1990). A review of pseudorandom number generators. *Computer Physics Communications*, 60(3):329–344. DOI: 10.1016/0010-4655(90)90032-V.

[Jamil et al., 2016] Jamil, M. A., Arif, M., Abubakar, N. S. A., and Ahmad, A. (2016). Software testing techniques: A literature review. In *6th International Conference on Information and Communication Technology for The Muslim World*, pages 177–182. DOI: 10.1109/ICT4M.2016.045.

[Janicic and Tošić, 2008] Janicic, M. V. and Tošić, D. (2008). The role of programming paradigms in the first programming courses. *The Teaching of Mathematics*, 11:63–83.

[Klammer and Kern, 2015] Klammer, C. and Kern, A. (2015). Writing unit tests: It's now or never! In *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation Workshops, 2015*, pages 1–4. DOI: 10.1109/ICSTW.2015.7107469.

[Körner et al., 2022] Körner, P., Leuschel, M., Barbosa, J., Costa, V. S., Dahl, V., Hermenegildo, M. V., Morales, J. F., Wielemaker, J., Diaz, D., Abreu, S., and Ciatto, G. (2022). 50 years of prolog and beyond. *Computing Research Repository*, abs/2201.10816. DOI: 10.1017/S1471068422000102.

[Krook et al., 2023] Krook, R., Smallbone, N., Svensson, B. J., and Claessen, K. (2023). Quickercheck: Implementing and evaluating a parallel run-time for quickcheck. In *ACM International Conference Proceeding Series*, pages 9 – 11. DOI: 10.1145/3652561.3652570.

[Lampropoulos et al., 2017] Lampropoulos, L., Gallois-Wong, D., Hrictcu, C., Hughes, J., Pierce, B. C., and yao Xia, L. (2017). Beginner's luck: a language for property-based generators. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages, 2017*, pages 114–129. ACM. DOI: 10.1145/3009837.3009868.

[Lampropoulos et al., 2019] Lampropoulos, L., Hicks, M., and Pierce, B. C. (2019). Coverage guided, property based testing. *Proceedings of the ACM on Programming Languages*, 3. DOI: 10.1145/3360607.

[Lampropoulos et al., 2018] Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. (2018). Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2. DOI: 10.1145/3158133.

[Lempel et al., 2007] Lempel, R., Mass, Y., Ofek-Koifman, S., Sheinwald, D., Petruschka, Y., and Sivan, R. (2007). Just in time indexing for up to the second search. In *Just in time indexing for up to the second search*, page 97–106, New York, NY, USA. Association for Computing Machinery. DOI: 10.1145/1321440.1321457.

[Lo et al., 2019] Lo, F.-Y., Chen, C.-H., and Chen, Y.-P. (2019). Genetic algorithms as shrinkers in property-based testing. In *Genetic and Evolutionary Computation Conference Companion, 2019*, page 291 – 292. DOI: 10.1145/3319619.3322004.

[Lo et al., 2020] Lo, F.-Y., Chen, C.-H., and Chen, Y.-P. (2020). Shrinking counterexamples in property-based testing with genetic algorithms. In *IEEE Congress on Evolutionary Computation, 2020*. DOI: 10.1109/CEC48606.2020.9185807.

[Löscher and Sagonas, 2017] Löscher, A. and Sagonas, K. (2017). Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2017*, page 46 – 56. DOI: 10.1145/3092703.3092711.

[Löscher and Sagonas, 2018] Löscher, A. and Sagonas, K. (2018). Automating targeted property-based testing. In *IEEE 11th International Conference on Software Testing, Verification and Validation, 2018*, page 70 – 80. DOI: 10.1109/ICST.2018.00017.

[MacIver and R., 2019] MacIver and R., D. (2019). Hypothesis: Property-based testing in python. *Journal of Open Source Software*, 4(43):1891. DOI: 10.21105/joss.01891.

[MacIver and Donaldson, 2020] MacIver, D. R. and Donaldson, A. F. (2020). Test-case reduction via test-case generation: Insights from the hypothesis reducer. In *Leibniz International Proceedings in Informatics*, volume 166. DOI: 10.4230/LIPIcs.ECOOP.2020.13.

[Mista and Russo, 2021] Mista, A. and Russo, A. (2021). Deriving compositional random generators. In *31st Symposium on Implementation and Application of Functional Languages, 2021*. ACM. DOI: 10.1145/3412932.3412943.

[Myers, 1979] Myers, G. J. (1979). *The Art of Software Testing*. John Wiley & Sons, New York.

[Nagy and Alvaro, 2022] Nagy, S. and Alvaro, P. (2022). The fun in fuzzing. *Queue*, 20:80 – 87. DOI: 10.1145/3580504.

[Nelson et al., 2021] Nelson, T., Rivera, E., Soucie, S., Del Vecchio, T., Wrenn, J., and Krishnamurthi, S. (2021). Automated, targeted testing of property-based testing predicates. *The Art, Science, and Engineering of Programming*, 6(2). DOI: 10.22152/programming-journal.org/2022/6/10.

[Padhye et al., 2019] Padhye, R., Lemieux, C., Sen, K., Papadakis, M., and Traon, Y. L. (2019). Validity fuzzing and parametric generators for effective random testing. In *Proceedings of the IEEE 41st International Conference on Software Engineering: Companion, 2019*, page 266 – 267. DOI: 10.1109/ICSE-Companion.2019.00107.

[Paraskevopoulou et al., 2015] Paraskevopoulou, Z., Hritçu, C., Dénès, M., Lampropoulos, L., and Pierce, B. C. (2015). Foundational property-based testing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9236:325 – 343. DOI: 10.1007/978-3-319-22102-1_22.

[Pike, 2014] Pike, L. (2014). Smartcheck: Automatic and efficient counterexample reduction and generalization. *ACM Special Interest Group on Programming Languages*, 49:53 – 64. DOI: 10.1145/2633357.2633365.

[Quadri and Farooq, 2010] Quadri and Farooq, S. U. (2010). Software testing – goals, principles, and limitations. *International Journal of Computer Applications*, 6. DOI: 10.5120/1343-1448.

[Robinson, 1965] Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of ACM*, 12(1):23–41. DOI: 10.1145/321250.321253.

[Runciman et al., 2008] Runciman, C., Naylor, M., and Lindblad, F. (2008). Smallcheck and lazy smallcheck: Automatic exhaustive testing for small values. In *Haskell'08 - Proceedings of the ACM 2008 Haskell Symposium*, page 37 – 48. DOI: 10.1145/1411286.1411292.

[Samuel, 2018] Samuel, M. (2018). An insight into programming paradigms and their programming languages. In *Proceedings of the Journal of Applied Technology and Innovation*, volume 1, pages 37 – 57.

[Sharma et al., 2021] Sharma, A., Demir, C., Ngomo, A. C. N., and Wehrheim, H. (2021). Mlcheck- property-driven testing of machine learning models. *Computing Research Repository*, abs/2105.00741. DOI: 10.1109/ICMLA52953.2021.00123.

[Sharma et al., 2022] Sharma, A., Melnikov, V., Hullermeier, E., and Wehrheim, H. (2022). Property-driven testing of black-box functions. In *IEEE/ACM 10th International Conference on Formal Methods in Software Engineering, 2022*, page 113 – 123. DOI: 10.1145/3524482.3527657.

[Sinha and Smidts, 2006] Sinha, A. and Smidts, C. (2006). An experimental evaluation of a higher-ordered-typed-functional specification-based test-generation technique. *Empirical Software Engineering*, 11:173–202. DOI: 10.1007/s10664-006-6401-9.

[Song et al., 2019] Song, D., Lee, M., and Oh, H. (2019). Automatic and scalable detection of logical errors in functional programming assignments. *Proceedings of the ACM on Programming Languages*, 3:1 – 30. DOI: 10.1145/3360614.

[Stanley, 2017] Stanley, J. (2017). Hedgehog will eat all your bugs. *Available online at: https://hedgehog.qa/*. last accessed in October 2024.

[Sterling and Shapiro, 1986] Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MIT Press, Cambridge, MA.

[Tarjan, 1971] Tarjan, R. (1971). Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 114–121. DOI: 10.1109/SWAT.1971.10.

[Umar, 2020] Umar, M. A. (2020). A study of software testing: Categories, levels, techniques, and types. *System Research and Information Technologies*, pages 42–52. DOI: 10.36227/techrxiv.12578714.v1.

[Vasconcelos and Ribeiro, 2020] Vasconcelos, P. and Ribeiro, R. P. (2020). Using property-based testing to generate feedback for c programming exercises. In *Proceedings of the OpenAccess Series in Informatics*, volume 81, pages 5 – 11. DOI: 10.4230/OASIcs.ICPEC.2020.28.

[Verma et al., 2017] Verma, A., Khatana, A., and Chaudhary, S. (2017). A comparative study of black box testing and white box testing. *International Journal of Computer Sciences and Engineering*, 5:301–304. DOI: 10.26438/ijcse/v5i12.301304.

[Warren, 1983] Warren, D. H. (1983). An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, Menlo Park, CA, USA.

[Warren et al., 2023] Warren, D. S., Dahl, V., Eiter, T., Hermenegildo, M. V., Kowalski, R. A., and Rossi, F., editors (2023). *Prolog: The Next 50 Years*, volume 13900 of *Lecture Notes in Computer Science*. Springer. DOI: 10.1007/978-3-031-35254-6.

[Wielemaker et al., 2010] Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2010). Swiprolog. *CoRR*, abs/1011.5332. DOI: 10.1017/S1471068411000494.

[Wrenn et al., 2020] Wrenn, J., Nelson, T., and Krishnamurthi, S. (2020). Using relational problems to teach property-based testing. *The Art, Science, and Engineering of Programming*, 5:20 – 23. DOI: 10.1145/3594671.

[Ziat et al., 2021] Ziat, G., Dien, M., and Botbol, V. (2021). Automated random testing of numerical constrained types. In *Leibniz International Proceedings in Informatics, 2021*, volume 210. DOI: 10.4230/LIPIcs.CP.2021.59.

Appendix A

Unit Tests Code

A.1 Exercise *dec2bin*

```
pfl_same_list_or_rev(L,L):- !.
pfl_same_list_or_rev(L,Rev):-
    reverse(L,Rev).

pfl_dec2bin_aux(0, []):- !.
pfl_dec2bin_aux(Dec, [Bit|T]):-
        Bit is Dec mod 2,
        Next is Dec div 2,
        pfl_dec2bin_aux(Next, T).

pfl_dec2bin(Dec, List, N):-
        Dec >= 0,
        pfl_dec2bin_aux(Dec, Aux),
        reverse(Aux, Temp),
        length(Temp, Used),
        Needed is N - Used,
        Needed >= 0,
        length(Aux2, Needed),
        maplist( =(0), Aux2),
        append(Aux2, Temp, List).

dec2bin(6, L, 0).       % 0 bits
dec2bin(6, L, 8).       % simple example
dec2bin(6, _L, 8), length(_L,N).        % simple example, length only

% simple example, accepts inverted
dec2bin(6, _L, 8), !, pfl_dec2bin(6,_L2,8), pfl_same_list_or_rev(_L,_L2).
% simple, with bigger list
dec2bin(6, L, 16), write(L).
```

```
% simple example, with bigger list, length only
dec2bin(6, L, 16), length(_L,N).

% simple example, with bigger list, accepts inverted
dec2bin(6, _L, 16),!, pfl_dec2bin(6,_L2,16), pfl_same_list_or_rev(_L,_L2).

dec2bin(6, L, 4).       % simple, with smaller list
dec2bin(14, L, 8).      % another example, 8
dec2bin(14, L, 16), write(L).   % another example, 16
dec2bin(14, _L, 16), length(_L,N).      % another example, 16, length only

% another example, 16, accepts inverted
dec2bin(14, _L, 16), !, pfl_dec2bin(14,_L2,16),
    pfl_same_list_or_rev(_L,_L2).

dec2bin(14, L, 4).      % another example, 4
dec2bin(0, L, 8).       % zero
dec2bin(1, L, 8).       % one
dec2bin(255, L, 8).     % full
dec2bin(15, L, 4).      % full, small
dec2bin(38, L, 4).      % fail
dec2bin(836, L, 8).     % fail
```

A.2 Exercise *matches*

```
matches([[3,6,8],[5,0,7],[2,1,4]],[3,5,2]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]],[1,5,9,13]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]],[2,6,10,14]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]],[5,10,2,23]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]],[1,5,9,13,17]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12]],[1,5,9]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12]],[2,6,10]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12]],[5,10,2]).
matches([[1,2,3,4],[5,6,7,8],[9,10,11,12]],[1,5,9,13]).
```

A.3 Exercise *next_gen*

```
:- dynamic(rule/2).

dec2bin_aux(0, []):- !.
dec2bin_aux(Dec, [Bit|T]):-
        Bit is Dec mod 2,
```

```
        Next is Dec div 2,
        dec2bin_aux(Next, T).

dec2bin(Dec, List, N):-
        Dec >= 0,
        dec2bin_aux(Dec, Aux),
        reverse(Aux, Temp),
        length(Temp, Used),
        Needed is N - Used,
        Needed >= 0,
        length(Aux2, Needed),
        maplist( =(0), Aux2),
        append(Aux2, Temp, List).

update_rule(N):-
        dec2bin(N, Bits, 8),
        abolish(rule/2),
        between(0,1,FirstBit),
        between(0,1,SecondBit),
        between(0,1,ThirdBit),
        Index is 8 - (FirstBit * 4 + SecondBit * 2 + ThirdBit),
        nth1(Index, Bits, Bit),
        assert(rule(FirstBit-SecondBit-ThirdBit, Bit)),
        fail.
update_rule(_).

update_rule(210), next_gen([0,0,1,0,1,1,1,0], NG).
next_gen([0,0,0,1,0,0,1,1,1,1,0,0,1,1,0,0], NG).
next_gen
    ([1,0,0,1,0,0,1,1,0,0,1,1,1,0,0,0,0,1,0,1,0,1,0,1,0,0,0,1,1,1,0,1], NG)
    .
update_rule(45), next_gen([0,0,1,0,1,1,1,0], NG).
next_gen([0,0,0,1,0,0,1,1,1,1,0,0,1,1,0,0], NG).
next_gen
    ([1,0,0,1,0,0,1,1,0,0,1,1,1,0,0,0,0,1,0,1,0,1,0,1,0,0,0,1,1,1,0,1], NG)
    .
```

A.4 Exercise *largest_circle*

```
circle_size(Person, Size):-
        collect([Person], People),
        length(People, Size).


```

```
collect( [H|T], People):-
        gives_gift_to(H, _, N),
        \+ member(N, [H|T]), !,
        collect( [N,H|T], People).
collect(People, People).

% gives_gift_to(Giver, Gift, Receiver)

gives_gift_to(marciliano, 'Those in Peril', nivaldo).
gives_gift_to(nivaldo, 'Vicious Circle', sandrino).
gives_gift_to(sandrino, 'Predator', marciliano).
gives_gift_to(bernardo, 'The Exchange', celeste).
gives_gift_to(celeste, 'The Brethren', desidero).
gives_gift_to(desidero, 'The Summons', felix).
gives_gift_to(felix, 'River God', juvenal).
gives_gift_to(juvenal, 'Seventh Scroll', leon).
gives_gift_to(leon, 'Sunbird', mario).
gives_gift_to(mario, 'Other book', bernardo).

gives_gift_to(newguy, 'New Book', newgal).
gives_gift_to(newgal, 'New Book 2', othergal).
gives_gift_to(othergal, 'New Book 3', otherguy).
gives_gift_to(otherguy, 'New Book 4', newguy).
gives_gift_to(one, 'New 1', two).
gives_gift_to(two, 'New 2', three).
gives_gift_to(three, 'New 3', four).
gives_gift_to(four, 'New 4', five).
gives_gift_to(five, 'New 5', six).
gives_gift_to(six, 'New 6', seven).
gives_gift_to(seven, 'New 7', one).

% Number of results (2)
findall(P, largest_circle(P), _L), length(_L, NResults).

% two groups of six, Sizes (6, 6)
largest_circle(_People), length(_People, Size).

% two groups of six, Sizes (6, 6), first output only
largest_circle(_People), !, length(_People, Size).

% two groups of six, Names (ordered lists)
largest_circle(_People), sort(_People, Sorted).

% two groups of six, Names (ordered lists), first output only
largest_circle(_People), !, sort(_People, Sorted).
```

```
51
52 % one group of seven, Sizes (7)
53 retract(gives_gift_to(seven, 'New 7', one)), assert(gives_gift_to(seven, '
       New 7', eight)), assert(gives_gift_to(eight, 'New 8', one)), findall(P,
        largest_circle(P), _L), length(_L, NResults), largest_circle(_People),
        length(_People, Size).
54
55 % one group of seven, Sizes (7), first output only
56 largest_circle(_People), !, length(_People, Size).
57
58 % one group of seven, Names (ordered list)
59 largest_circle(_People), sort(_People, Sorted).
60
61 % one group of seven, Names (ordered list), first output only
62 largest_circle(_People), !, sort(_People, Sorted).
```