

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Property-Based Testing Input Generation for Prolog using Constraint Programming

Henrique Filipe Pereira da Silva Caridade



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 17, 2026

Property-Based Testing Input Generation for Prolog using Constraint Programming

Henrique Filipe Pereira da Silva Caridade

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

President: Prof. Alexandra Sofia Ferreira Mendes

Referee: Prof. Jorge Manuel Neves Coelho

Referee: Prof. António Mário da Silva Marcos Florido

July 17, 2026

Resumo

Property-Based Testing (**PBT**) constitui uma técnica de teste de software que melhora a cobertura dos testes através da geração automática de casos de teste e da verificação de propriedades gerais, em vez de casos de teste individuais, por exemplo, Unit Testing (**UT**). Contudo, a geração de entradas válidas e estruturalmente complexas continua a ser um desafio significativo, particularmente em Prolog, onde as aplicações dependem frequentemente de estruturas de dados complexas e de restrições relacionais. Constraint Logic Programming (**CLP**) oferece uma abordagem declarativa para expressar e resolver estas restrições, constituindo uma base promissora para melhorar a geração de dados de teste.

Esta dissertação investiga a utilização de **CLP** para suportar a geração eficiente de entradas num framework de Property-Based Testing para SICStus Prolog. Numa primeira fase, é realizada uma revisão do estado da arte das técnicas de geração de dados baseadas em **CLP**, identificando as abordagens mais adequadas ao teste de caixa negra e evidenciando a escassez de investigação na área da geração aleatória baseada em restrições para Property-Based Testing. Com base nesta análise, é proposta uma extensão de um framework de **PBT** existente para Prolog, através da implementação de uma estratégia de geração de bases de dados por etapas, recorrendo a **CLP** para construir entradas estruturadas válidas e consistentes, assegurando simultaneamente o cumprimento das restrições de integridade ao longo de todo o processo de geração.

A abordagem proposta é avaliada através de um caso de estudo no contexto da avaliação automática de exercícios de programação em Prolog e comparada com uma abordagem tradicional baseada em **UT**. Os resultados demonstram que a utilização de **CLP** constitui um mecanismo eficaz e expressivo para a geração de entradas estruturadas complexas, permitindo testar cenários cuja criação manual seria significativamente mais difícil. Embora a comparação experimental com **UT** não permita retirar conclusões definitivas quanto à eficácia global de ambas as abordagens, o framework desenvolvido alarga com sucesso a aplicabilidade do **PBT** a problemas dependentes de bases de dados e de estruturas de dados complexas, mantendo uma arquitetura declarativa e extensível.

Esta dissertação contribui tanto com uma revisão abrangente das técnicas de geração de dados baseadas em **CLP** como com uma extensão prática de um framework de **PBT** para Prolog, estabelecendo uma base para trabalho futuro nas áreas da geração dirigida de entradas, estratégias de shrinking e definição de propriedades mais expressivas.

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

Abstract

Property-Based Testing (**PBT**) is an effective software testing technique that improves test coverage by automatically generating inputs and verifying general properties instead of individual test cases, like, e.g., Unit Testing (**UT**). However, generating valid and structurally complex inputs remains a significant challenge, particularly in Prolog, where applications often rely on intricate data structures and relational constraints. Constraint Logic Programming (**CLP**) offers a declarative approach to expressing and solving such constraints, making it a promising foundation for improving input generation.

This dissertation investigates the use of **CLP** to support efficient input generation for a **PBT** framework in SICStus Prolog. It begins with a review of the state of the art in **CLP**-based data generation techniques, identifying approaches suitable for black-box testing and highlighting the limited research on constraint-based random generation for **PBT**. Based on this analysis, the dissertation extends an existing Prolog **PBT** framework by implementing a stage-based database generation strategy that uses **CLP** to construct valid and consistent structured inputs while enforcing integrity constraints throughout the generation process.

The proposed approach is evaluated through a case study involving automatic assessment of Prolog programming assignments and compared with a traditional **UT** approach. The results demonstrate that **CLP** provides an effective and expressive mechanism for generating complex structured inputs, enabling the testing of scenarios that would be difficult to create manually. Although the experimental comparison with **UT** is not sufficient to draw definitive conclusions regarding overall effectiveness, the proposed framework successfully broadens the applicability of **PBT** to database-dependent and structurally complex problems while maintaining a declarative and extensible design.

The dissertation contributes both a comprehensive review of **CLP**-based input generation techniques and a practical extension to an existing **PBT** framework for Prolog, laying the groundwork for future research on targeted input generation, shrinking strategies, and more expressive property definitions.

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

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Problem Definition	3
1.4	Objectives	3
1.5	Contributions	4
1.6	Dissertation Structure	4
2	Fundamentals of Software Testing, Logic Programming and CLP	5
2.1	Software Testing	5
2.1.1	White-box Testing vs Black-box Testing	7
2.1.2	Unit Testing	7
2.1.3	Property-Based Testing	9
2.2	Prolog and Constraint Logic Programming	11
2.2.1	Logic Programming	11
2.2.2	Prolog	12
2.2.3	Constraint Logic Programming	14
2.2.4	CLP in Prolog	15
2.3	Summary	18
3	Data Generation with CLP	19
3.1	Bounded-Exhaustive Testing	20
3.1.1	Example of CLP generator for Red-Black Trees	22
3.2	Constraint-Based Random Testing	25
3.2.1	Rejection Sampling	25
3.2.2	Constraint-Based Generators	25
3.2.3	Graph Representation of a CSP for Random Sampling	28
3.3	Summary	29
4	Methodology	31
4.1	Proposed Solution and Architecture	31
4.2	Planning and Work Organization	31
4.2.1	Theoretical Analysis	31
4.2.2	Development	32
4.3	Experimental Process and Evaluation Design	32
4.4	Expected Results and Relevance	32
4.5	Risk Analysis	32
4.6	Summary	33

5	Development of Database Generation for a PBT Library with CLP	34
5.1	Main Architecture	34
5.2	CLP for Basic and Composite Data Types	35
5.2.1	Basic Types	36
5.2.2	Composite Types	38
5.2.3	Constrainer	39
5.3	Database Generation with CLP	40
5.3.1	Final Implementation	40
5.3.2	Development Process	43
5.4	Generator	46
5.5	Testing Interface	48
5.6	Limitations	50
5.7	Summary	51
6	Experiment and Results	53
6.1	Setup	53
6.2	Database Example	54
6.2.1	Books and Authors Database	54
6.2.2	Similarity Experiment	57
6.3	Results and Conclusions	59
6.4	Summary	60
7	Conclusion	61
7.1	Summary of Findings	61
7.2	Main Contributions	62
7.3	Future Work	62
	Bibliography	63
A	Data Generation with CLP Queries	66

List of Figures

2.1	Every software project has optimal test effort [Jamil et al., 2016]	6
2.2	Backtracking with DFS strategy [Apt, 1997]	13
4.1	Gantt Chart for the Work Plan	32
5.1	Updated Architecture of the developed PBT Library	35
5.2	Flow Graph of the Main Testing Workflow	35
6.1	Overview of the Python testing library workflow (extracted from [de Sá, 2025])	54

List of Tables

3.1	Quantitative results of queries on Data Generation with CLP	20
4.1	Risks and mitigation strategies	33
6.1	Distribution of submission types	59
6.2	Summary of evaluation statistics	59
A.1	Queries on Data Generation with CLP	67

Listings

2.1	Unit testing using pytest framework [de Sá, 2025]	8
2.2	Property definition using QuickCheck framework [Claessen and Hughes, 2000]	9
2.3	Example of a successful QuickCheck test [Claessen and Hughes, 2000]	10
2.4	Wrong prop_RevApp property definition [Claessen and Hughes, 2000]	10
2.5	Result of wrong prop_RevApp property [Claessen and Hughes, 2000]	10
2.6	Joining 2 variables in a list with Unification	12
2.7	Program example for backtracking. Adapted from [Apt, 1997]	13
2.8	SICStus Prolog Constraint Logic Programming (CLP) libraries	15
2.9	Sudoku solver with CLP in SICStus Prolog	16
2.10	Sudoku solver example query result	17
2.11	Sudoku solver without line 11 query result	17
3.1	Template for developing test case generators [Fioravanti et al., 2015]	22
3.2	Red Black Tree Definition	22
3.3	Red Black Tree Definition	23
3.4	Red Black Tree Shape Definition	23
3.5	Red Black Tree Path Invariant (I_1)	24
3.6	Red Black Tree Color Invariant (I_2)	24
3.7	Red Black Tree Ordered Invariant (I_3)	24
5.1	Example of arbitrary constraints	36
5.2	oneof arbitrary constraint	36
5.3	listof arbitrary constraint	37
5.4	Name and Country arbitraries	38
5.5	Example of composite constraints	39
5.6	Constrain arguments function	39
5.7	Books and Authors Database Example	40
5.8	Database Predicate Parser	42
5.9	Simple Database Example	43
5.10	Composite Types for Books and Authors Database	43
5.11	Books and Authors Database Naive Example	44
5.12	Constrained Length Predicate	44
5.13	All Distinct for Constrained Length Lists Predicate	45
5.14	CSPs Divider	46
5.15	Rejection Sampling	47
5.16	Uniform Rejection Sampling	47
5.17	Random Variable Value Predicate	48
5.18	Property Example	48
5.19	Custom arbitrary Example	49
5.20	Custom Database Example	49

5.21 Quickcheck Predicate Examples	49
5.22 Input/Output Predicate Example	50
5.23 Database Arguments Example	51
6.1 Books and Authors Composite Types	55
6.2 Books and Authors Database Example	55
6.3 Books and Authors Database for Unit Tests Example	56
6.4 Similarity Example Solution	57
6.5 Books and Authors Generic Property	58
6.6 Books and Authors Unit Test Suite	58

List of Acronyms

BET	Bounded-Exhaustive Testing
CC	Connected Component
CLP	Constraint Logic Programming
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
FEUP	Faculty of Engineering of the University of Porto
LP	Logic Programming
PBT	Property-Based Testing
TPBT	Targeted Property-Based Testing
UT	Unit Testing
UV	Utility Value

Chapter 1

Introduction

Software development is one of the most important fields of technological advancement today. Many new technologies are not purely mechanical; more often than not, there is a chip, a micro-processor, or a full-fledged processor that needs software to work correctly and reliably. For that to be possible, software developers had to create ways to guarantee good performance, correctness, and reliability of their creations, so software testing was created. Software testing is becoming increasingly more important as software is being used more and in more critical situations.

This chapter introduces the context for this dissertation, the motivation for its creation, the definition of the problem to be solved, the objectives to be achieved, and lastly, the contributions of this work to the field of software testing as a whole. Also, the last section describes the overall document structure.

1.1 Context

Software Testing has a critical role in software development nowadays, verifying software quality factors like reliability, usability, integrity, security, etc., of a program or application. It has an imperative role in areas such as aerospace engineering, vehicle software, medicine-related software, and other areas where any errors are not acceptable; the software must work 100% of the time, otherwise people's lives are in danger. Other than those critical areas, it's still an objective to reduce the number of errors in any software to improve its correctness, performance, and reliability; that's why software testing has become a vital part of the software lifecycle for any piece of software created nowadays.

Unit Testing (**UT**) is the most common type of software testing used. It works very well because it's intuitive and it's easy to implement multiple unit tests, but it has its shortcomings. When we have a problem with a wide range of possible inputs, it's challenging with unit tests to cover all of them effectively in a way that covers all possible errors, and it doesn't take an enormous amount of time and code to write those tests. For those kinds of problems, a better approach was devised: Property-Based Testing (**PBT**).

PBT was initially introduced and popularized by QuickCheck [Claessen and Hughes, 2000] for the 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 Hypothesis¹ in Python, PropEr² in Erlang, or PrologCheck in Prolog [Amaral et al., 2014].

PBT works by instead of defining what the correct output for a certain input (like with Unit Testing), we define properties that must hold for the outputs for a certain kind/range of different inputs. The **PBT** testing tool then automatically generates any number of different inputs and checks if the properties hold. This offers a couple of advantages in contrast with the usual Unit Testing approach. First, it takes much less code to write properties than it takes to write multiple unit tests for the same coverage of errors. And second, it covers way more possibilities that would be unfeasible to write all by hand, creating better code coverage and an overall more robust testing system.

Another great advantage of **PBT** is the automatic shrinking process. When the **PBT** tool finds an input that breaks a property, it tries to simplify the input to its simplest form that still breaks the same property. This is very useful for the software developers working on the code under test because it provides the simplest input that breaks, giving them a better chance at easily finding the error and rectifying it without getting lost in the complexity of the input when it isn't what causes the error.

One of the problems a **PBT** tool implementation has to solve is the data generation problem; two problems are important in this context: First, we must decide whether it generates data completely at random or if it is targeted. Second, we must find a way to accommodate every possible input; it could simply be an integer, or it could need (in the case of Prolog) the database to be pre-populated with some input data.

This dissertation is a direct continuation of the work done by a fellow Faculty of Engineering of the University of Porto (**FEUP**) student on the implementation of a **PBT** tool for Prolog [de Sá, 2025].

1.2 Motivation

Property-Based Testing is a powerful approach to Software Testing that any language can benefit from because its core idea has no ties to one language. As a result of this abstraction, software testers can purely think about the properties the system under test should have, instead of thinking about how to implement the tests as code, leading to more high-level property definitions that are more readable and maintainable. With that being said, in the context of Prolog, there aren't many **PBT** tools available; the lack of availability of those tools for Prolog is one of the motivators for this work.

¹ Available online at: <https://hypothesis.readthedocs.io/en/latest/> (last access on June, 2026)

² Available online at: <https://github.com/proper-testing/proper> (last access on June, 2026)

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. 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.

1.3 Problem Definition

In the pursuit of an efficient input data generation process in the context of Property-Based Testing in Prolog, we encounter two challenges: Targeted input data generation and Generating Data Structures/Populating Databases efficiently.

Targeted Property-Based Testing (**TPBT**) [Löschner and Sagonas, 2017] is a sub-branch of Property-Based Testing where the input data is not totally random but instead is biased into ranges, values, or internal structures of the input data that are critical to the function under test. This is very useful in more complicated data structures, for example, with graphs where the internal structure of a graph might be what matters most for the function under test, but with random input generation, most of the tests will be with an average connected graph, and the test might give a false positive result. Of course, this could be solved simply by cranking up the number of input tests we are creating, reducing the possibility of a false positive, but that will tank the performance of the testing tool. That's why **TPBT** is a very interesting option to consider to make a more robust testing tool.

Another problem we have is the actual generation of the data structures when they are complicated and intricate. For this problem, Constraint Logic Programming (**CLP**) [Jaffar and Lassez, 1987] is a very interesting tool that could simplify this process immensely. **CLP** is a form of Logic Programming (**LP**) that contains constraints in the body of clauses, that means that instead of trying to prove a goal by backtracking every time a sub-goal fails, the interpreter saves the constraints in a set, and then a solver tries to tell if the constraint set is satisfiable or not. This could be used in data generation using the **CLP** solver to cut down invalid options in scenarios where guaranteeing the data integrity and consistency upon generation is challenging, for example, in a database where we have a Book table and Author table, the relationship between a book and an author must be valid, i.e. a book must be created on a date after its author was born.

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 these in Prolog efficiently, while adhering to programming language best practices. The research presented throughout this document addresses the following research questions:

- **RQ1:** How to do input generation for **PBT** with complex data structures?

- **RQ2:** Is **CLP** useful for input generation for **PBT**?
- **RQ3:** How does **PBT** with **CLP** in Prolog, compare to traditional **UT** in academic automatic grading systems?

To answer these questions, a theoretical analysis of the state-of-the-art techniques for structured data generation with **CLP** was conducted, and the development of an extension to the existing Prolog tool will be conducted.

1.5 Contributions

This dissertation contributes to improving Property-Based Testing (**PBT**) in Prolog by addressing efficient input data generation using Constraint Logic Programming (**CLP**). The main contributions are:

- The design and implementation of a **CLP**-based input generation framework for complex data structures in Prolog;
- An experimental evaluation comparing the proposed approach with traditional **UT** strategies;
- An extension of the existing Prolog **PBT** tool [de Sá, 2025].

1.6 Dissertation Structure

The dissertation is structured into 6 more chapters. Chapter 2 presents the theoretical foundations of software testing, Property-Based Testing (**PBT**), Prolog, and Constraint Logic Programming (**CLP**). Chapter 3 reviews the state of the art in data generation with **CLP**, focusing on techniques suitable for **PBT**. Chapter 4 describes the proposed methodology and the architecture of the **CLP**-based generation framework. Chapter 5 details the implementation of the database generation engine and its integration into the Prolog **PBT** framework. Chapter 6 presents the experimental evaluation and compares the proposed approach with traditional Unit Testing (**UT**). Finally, Chapter 7 concludes the dissertation by summarising the main findings, contributions, and future work.

Chapter 2

Fundamentals of Software Testing, Logic Programming and CLP

This chapter introduces the fundamental concepts required to understand the techniques presented in the remainder of this thesis. More specifically, it provides an overview of software testing, Property-Based Testing (PBT), Logic Programming (LP), and Constraint Logic Programming (CLP). These concepts form the theoretical foundation for Targeted Property-Based Testing (TPBT) and for CLP as a mechanism for automated test data generation in Prolog, which is what this thesis focuses on.

This chapter is structured into 2 sections. The first section provides an overview of software testing, distinguishing White-box Testing vs Black-box Testing, and presents Unit Testing (UT) and Property-Based Testing (PBT). The second section offers an introduction to Logic Programming and CLP and introduces Prolog and its CLP capabilities.

2.1 Software Testing

Software testing is a fundamental piece of the modern software development lifecycle, and it plays a crucial role in ensuring software quality. As software systems are being increasingly adopted by critical domains such as healthcare and finance, the consequences of software failures have become more severe. Therefore, software testing is becoming more and more important to fight those failures by evaluating whether a system behaves according to its intended requirements.

At its core, software testing is the process of verifying and validating that a software application or program meets the business and technical requirements that guided its design and development and behaves as expected. It primarily aims to identify errors in the application under test, but it is also used to test software for other quality factors such as reliability, usability, integrity, security, efficiency, etc. [Quadri and Farooq, 2010]. Testing drastically reduces the chances of errors persisting in production. This is crucial because the cost to repair an error after the application's deployment is often exponentially higher [Quadri and Farooq, 2010].

Although software testing is a crucial and essential part of modern software development, its use 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 [Garousi et al., 2020]. This is because, in most software systems, the sheer number of combinations of input and state of a program is so big that testing every possibility would not be feasible. It wouldn't be feasible because modern computers are still not able to process at the speed that it would take to test all possibilities of some of the more complicated software systems in a reasonable amount of time, and even if they could, the cost of all of that processing would also be great. This limitation led to the development of clever testing strategies that make the most of the processing power available in a modern machine. These strategies can be broadly categorized into two families: White-box Testing and Black-box Testing, with several subtypes under each (also known as Open-box Testing and Closed-box Testing, respectively). These categorize whether or not a test suite has knowledge of the underlying source code of the application under test, respectively. There are various types of software testing, each with a specific goal and dependent on the type of application under test [Umar, 2020].

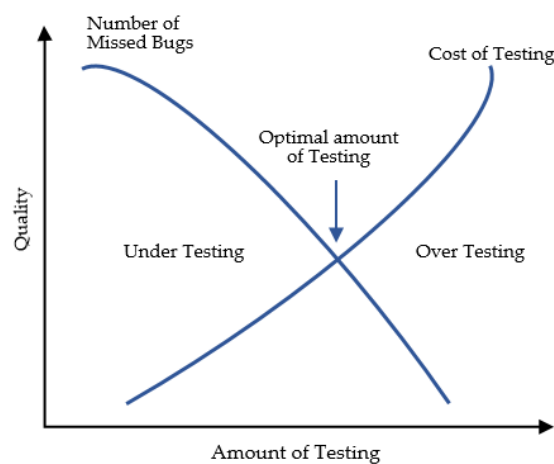


Figure 2.1: Every software project has optimal test effort [Jamil et al., 2016]

Although there is a cost associated with errors in production, there's also a cost associated with the creation of the tests themselves. We have to be careful to not over-test and incur more costs than we would have otherwise. This also highlights why we seek more efficient testing strategies that not only are efficient at finding errors but also efficient to create.

Despite there being multiple testing strategies, this dissertation focuses on only two of them: Unit Testing (UT) and Property-Based Testing (PBT). These testing approaches are chosen because PBT is the focus of this work, and UT is one of the most well-established strategies for software testing, and because of that, it's being used as a benchmark. The following sections start by explaining in more depth the difference between White-box Testing vs Black-box Testing and their subtypes, then explore more in depth the UT and the PBT strategies.

2.1.1 White-box Testing vs Black-box Testing

A testing strategy is classified as White-box Testing if the tester needs knowledge about the inner workings of a piece of software. It focuses on using test data that uses knowledge of the internal logic, code structure, and control flow of the application [Verma et al., 2017]. White-box Testing is divided into many different subtypes, such as Unit Testing, Integration Testing, Path Testing, Branch Testing, and Data Flow Testing.

Unit Testing (UT) is a broad term relating to any test that tests only a specific component or part of a component in isolation; this may include other testing techniques. UT is often paired with the concept of test coverage metrics. We can evaluate how well a test suite covers all of the possibilities by using different test coverage metrics. Examples of these code coverage metrics are:

- **Statement/line coverage:** percentage of statements/lines that a test suite has executed.
- **Branch coverage:** percentage of branch results that are explored by a test suite, counting both possible paths of a branch as a possibility.
- **Condition coverage:** percentage of individual boolean conditions outcomes that are tested, including the false and true results.

These metrics are designed to ultimately maximize the amount of the initial set of combinations of input and state that are tested and "tested" by proxy of other tests. Using UT with test coverage metrics is effective, but maintaining tests for ever-changing software can be costly because tests have to be constantly adjusted. One must be careful because even when we have 100% coverage in any or all of these metrics, it still doesn't guarantee that the unit tests are properly implemented or that the system will have no errors [Bissi et al., 2016].

On the other hand, a testing strategy is classified as Black-box Testing if the tester doesn't need knowledge about the inner workings of a piece of software; they only need to know the input and what is expected as output, as per the system's requirements [Verma et al., 2017]. Black-box Testing is also divided into many different subtypes, such as Property-Based Testing, Equivalence Partitioning Testing, Boundary Value Analysis, and Functional Testing.

Property-Based Testing (PBT) is a type of Black-box Testing that works by defining properties that should hold for the outputs of a certain set of inputs of a function. It then automatically generates random (or not totally random) test input data to see if the property holds true or not.

2.1.2 Unit Testing

Unit Testing (UT) is a software testing technique that consists of testing 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 is often very effective because it finds the root of the problem early, by having a test that covers that specific part of the code, preventing other parts of the code from affecting the outcome of the test.

UT's main advantages are its granularity and ease of use, causing errors to be detected early on. Because it aims to test small portions of code, when something inevitably fails, it's extremely easy to pinpoint where the issue is. Early error detection significantly reduces the cost of correcting them in later phases [Quadri and Farooq, 2010]. The fact that UT gives the input and output of a scenario turns the tests into automatic documentation for the proper use of the underlying code. Unit tests are easy to write, allowing less experienced developers to contribute effectively. One downside of UT is that sometimes its focus on specific parts of the code may cause it to not find problems related to the interactions between different components.

To implement unit tests, test frameworks are used that allow for the automation and efficient execution of tests, such as JUnit for Java and pytest 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 of different lengths.

```
1 import pytest
2
3 def test_reverse_simple():
4     assert reverse('hello') == 'olleh'
5
6 def test_reverse_empty_string():
7     assert reverse('') == ''
8
9 def test_reverse_single_character():
10    assert reverse('a') == 'a'
11
12 def test_reverse_2_characters():
13    assert reverse('ab') == 'ba'
14
15 def test_reverse_3_characters():
16    assert reverse('abc') == 'cba'
```

Listing 2.1: Unit testing using pytest framework [de Sá, 2025]

In this dissertation, UT is used as a benchmark due to its widespread integration in both industry and academic settings, as well as its effectiveness in detecting errors at the early stages of development [Bissi et al., 2016]. This facilitates a comprehensive analysis of how PBT complements or diverges from conventional practices, offering insights into its potential benefits and adoption barriers across different domains.

2.1.3 Property-Based Testing

Property-Based Testing (**PBT**) is a software testing technique that focuses on verifying general properties or invariants of the application under test, rather than validating its behavior by specifying correct input-output pairs. In contrast with **UT**, where you define each test input and corresponding output, **PBT** consists of stating properties that should hold for a range of inputs. After defining the properties, a **PBT** framework then automatically generates random inputs in the given range and tests if the outputs satisfy the properties [Claessen and Hughes, 2000].

The main idea behind **PBT** is that many software defects come from corner cases and unexpected combinations of inputs, instead of the typical straightforward ones. Manually identifying such cases is often difficult, particularly when dealing with large or complex input spaces. By automating the generation of test cases, **PBT** aims to explore these spaces and expose defects that might otherwise go under the radar.

PBT is a Black-box Testing technique because it operates solely on the inputs and outputs of the application under test, without requiring any knowledge of its implementation. **PBT** is most useful when you have a well-defined abstraction with a complex implementation [Goldstein et al., 2024]. Because of this, with **PBT**, you can still write efficient tests even without knowing the inner workings of the system.

The concept **PBT** started when researchers recognized that checking a system against overarching invariants might offer a more in-depth and trustworthy mechanism to guarantee the correctness of a system. The development of QuickCheck by Claessen and Hughes [2000] for the Haskell programming language marked a turning point in the development of **PBT**. They took two relatively old ideas, namely specifications as properties and random testing, and joined them to make the first **PBT** tool for software testing.

A property is commonly expressed as a predicate that takes one or more inputs and succeeds if the desired condition holds. Properties may encode invariants, algebraic laws, or consistency relations between different operations. Formally, a property can be seen as a universal predicate over the input domain, but in practice, **PBT** frameworks approximate that by executing the property over a large number of automatically generated inputs. While this does not guarantee correctness, it significantly increases confidence in the behaviour of the system under test when compared to traditional **UT** approaches.

Unlike the **UT** example in Listing 2.1, where test cases are discrete input-output pairs, in Listing 2.2, the properties defined validate invariants of the reverse function:

- reversing a single-element list preserves the element
- concatenating and reversing sublists produce consistent results
- double reversal restores the original list.

```

1 prop_RevUnit x = reverse [x] == [x]
2 prop_RevApp xs ys = reverse (xs++ys) == reverse ys++reverse xs

```

```
3 prop_RevRev xs = reverse (reverse xs) == xs
```

Listing 2.2: Property definition using QuickCheck framework [Claessen and Hughes, 2000]

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

```
1 Main> quickCheck prop_RevApp
2 OK: passed 100 tests.
```

Listing 2.3: Example of a successful QuickCheck test [Claessen and Hughes, 2000]

In Listing 2.3, the function `quickCheck` takes a property as a parameter and applies it to 100 randomly generated inputs and reports "OK" if the result was `True` in every case. If the property fails, then `quickCheck` reports the counterexample. For example, if we mistakenly define `prop_RevApp` (Listing 2.4), then checking the property will likely produce a result such as the one in Listing 2.5.

```
1 prop_RevApp xs ys = reverse (xs++ys) == reverse xs++reverse ys
```

Listing 2.4: Wrong `prop_RevApp` property definition [Claessen and Hughes, 2000]

```
1 Main> quickCheck prop_RevApp
2 Falsifiable, after 1 tests:
3 [2]
4 [-2,1]
```

Listing 2.5: Result of wrong `prop_RevApp` property [Claessen and Hughes, 2000]

Compared to **UT**, the automatic generation of hundreds of input examples makes it so that **PBT** offers a far wider coverage of the possible inputs, testing the system with inputs ranging from trivial to edge cases and unforeseen combinations. This variety and wide coverage provided by the generators make the result of the tests more robust and trustworthy.

Because the counterexamples found by random testing are usually complicated, barely comprehensible, they are often very hard use as a source to debug the underlying error. This is the reason shrinking is crucial and indispensable for **PBT** [Lo et al., 2019]. Shrinking reduces the counterexample found to its simplest form that still fails the property, making debugging easier.

Property-Based Testing has been evolving considerably, particularly with respect to input generation and identification of relevant properties. Traditional **PBT** relies on purely random input

generation, which can lead to part of the input space being left unchecked. Targeted Property-Based Testing (**TPBT**) addresses this limitation by incorporating search-based techniques that steer the generation of test inputs to more interesting and error-prone inputs. By applying strategies such as hill climbing and simulated annealing, **TPBT** systematically navigates the input space, increasing the chances of finding counterexamples [Löschner and Sagonas, 2017]. This guided approach is especially beneficial in scenarios where failures are rare and require complex input structures that are unlikely to be produced through random sampling alone. The way that **TPBT** evaluates different inputs with respect to how error-prone they are is by the use of Utility Values (**UVs**). **UVs** quantify how close a given input is to falsifying a property [Löschner and Sagonas, 2017]. These values act as feedback to the search process, enabling it to prioritize inputs with a higher **UV**.

Despite its advantages, it's still unclear whether **PBT** is superior to other methods, such as **UT**. Some of the disadvantages might be [Claessen et al., 2010]:

- it is often necessary to write test data generators for problem-specific data structures (other methods do not need to write generators at all).
- the developer must formulate a formal specification, which is conceptually more difficult than just predicting the correct output in specific examples.
- randomly generated test cases might potentially be less effective at revealing errors than carefully chosen ones.

A recent study on the use of **PBT** and **UT** by testers revealed that, as far as preference goes, testers didn't prefer one over the other [Goldstein et al., 2024]. Testers recognized both strategies as tools that should be available to the developers. When the code is simple enough, **UT** offers a lightweight option that gives the testers confidence. On the other hand, **PBT** becomes useful if the code is difficult to get right or when writing out enough examples becomes tedious.

2.2 Prolog and Constraint Logic Programming

This section introduces the programming paradigms that form the technical foundation of this dissertation. It starts by looking at the Logic Programming (**LP**) paradigm and introducing Prolog, followed by an overview of Constraint Logic Programming (**CLP**) as an extension that integrates constraint solving into **LP**, and it ends by analyzing Prolog and its support for **CLP**.

2.2.1 Logic Programming

Logic Programming (**LP**) is a declarative programming paradigm based on formal logic, in which programs are expressed as sets of logical statements, and computation is performed through logical inference. Rather than specifying a sequence of instructions, a logic program defines relations between entities, and execution consists of determining whether these relations hold for given queries.

In other words, as opposed to imperative paradigms where you define what to do, in **LP**, a program is constructed by describing what is true and making queries on that information.

"In traditional software engineering, one builds a program by specifying the operations to be performed in solving a problem, that is, by saying *how* the problem is to be solved. The assumptions on which the program is based are usually left implicit. In Logic Programming, one constructs a program by describing its application area, that is, by saying *what* is true. The assumptions are explicit, but the choice of operations is implicit." [Genesereth and Ginsberg, 1985]

In this view, in **LP** one doesn't just have a collection of procedures, but instead one defines a knowledge base encoding facts and rules about a certain domain. Queries posed to the program are logical questions, and computation consists of deriving answers that logically follow from the facts and rules previously defined in the program.

2.2.2 Prolog

Created in Marseilles in 1972, Prolog (name given by Philippe Roussel as an abbreviation for "PROgrammation en LOGique") was born of a project not aimed at producing a programming language but at processing natural languages [Colmerauer and Roussel, 1993]. The idea that eventually led to the creation of Prolog was to use natural language to communicate directly with a computer. Initially, Prolog was based on Horn clauses with some additions, but over time, more features were added to make the language not only more powerful and capable but also efficient [Warren et al., 2023]. Nowadays, Prolog is a highly sophisticated language and the most prominent implementation of the **LP** paradigm that serves as the base programming language for this dissertation.

At the heart of the underlying evaluation mechanism of Prolog lies unification. In **LP**, variables represent unknown values, and the values assigned to variables are terms (expressions). These values are assigned by means of certain substitutions, called most general unifiers. The process of computing most general unifiers is called unification [Robinson, 1965]. In a nutshell, unification is the act of taking two terms and comparing or making them equal by identifying identical substitutions between the two. The unification is a recursive process that makes sure that variables are bound to terms, and between terms, functors and arity or constants have to be equal. If the two terms can be unified, then unification is successful; otherwise, it fails. This process is the foundation for Prolog's evaluation mechanism because it provides pattern-matching capabilities when it succeeds, but also introduces the "negation as failure" concept when it fails.

```

1 join(X, Y, [X, Y]).
2
3 ?- join(a, b, Res).
4 Res = [a, b] ?

```

Listing 2.6: Joining 2 variables in a list with Unification

In Listing 2.6, we first define the predicate `join/3` which joins the first 2 arguments into a list. The query in that same listing attempts to unify the clause `join(a, b, Res)` with the predefined clause `join(X, Y, [X, Y])`. This causes the variable `X` to unify with the constant `a` and the variable `Y` to unify with the constant `b`, which then causes the variable `Res` to unify with the term `[X, Y]`, which is unified to `[a, b]` because both variables are already unified, giving the final answer `Res = [a, b]`.

In addition to unification, another extremely important component of the underlying evaluation mechanism of Prolog is backtracking. Backtracking allows the evaluation mechanism to visit all possibilities methodically to find a solution using a Depth-First Search strategy. For example, for the program in Listing 2.7, we have the resulting step-by-step search path in Figure 2.2.

```

1 p :- q.
2 p.
3 q :- r.
4 q :- s.
5 r :- fail.
6 s :- fail.

```

Listing 2.7: Program example for backtracking. Adapted from [Apt, 1997]

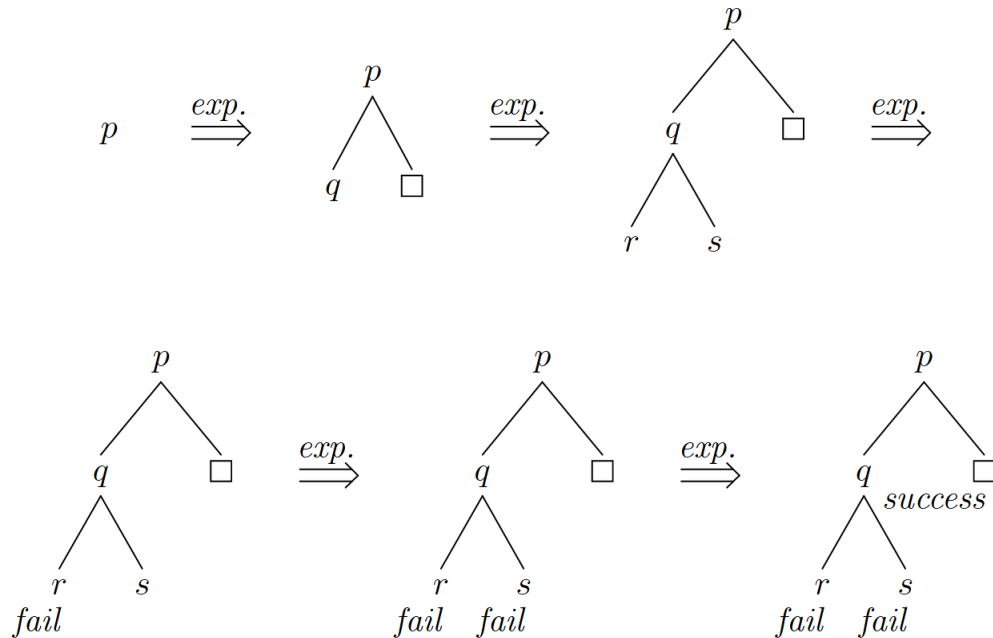


Figure 2.2: Backtracking with DFS strategy [Apt, 1997]

The Depth-First Search strategy consists of starting on the root node, in this case, the base query, and recursively going through the options of the nodes. If a leaf node leads to failure, then

the program backtracks to the last node with options left to explore and continues from there. If it finds a successful result, it returns the result. The order in which the definitions are traversed is the same as in the program, that's the reason p starts by going to the $p :- q.$ definition instead of the p in Listing 2.7 and Figure 2.2.

Prolog's simple syntax, paired with dynamic typing, makes the language easy to use and prototype with. Despite its advantages, Prolog has a clear drawback for the average developer, namely, most developers are used to imperative paradigm languages and not declarative paradigm languages, which creates a barrier to entry to the Logic Programming world. For a complete survey of logic programming, see [Apt, 1997].

In terms of the interpreters available, the most popular are SWI-Prolog¹ and SICStus² Prolog. SWI-Prolog was released in 1987, and despite gaining popularity primarily in education, its development has been driven by the needs of real-world applications. Now SWI-Prolog is neither a commercial Prolog system nor a purely academic enterprise but increasingly a community project [Wielemaker et al., 2010]. SICStus Prolog is more oriented to commercial use due to its superior performance and reliability on large systems. It also follows the standards set by the International Organization for Standardization (ISO), which only makes it more reliable, dependable, and suitable for large systems with proportionate amounts of data [Carlsson and Mildner, 2012].

2.2.3 Constraint Logic Programming

Constraint Logic Programming (CLP) extends Logic Programming by integrating constraint solving into the inference process. Instead of relying exclusively on unification and backtracking, CLP systems allow relations to be expressed as constraints over specific domains, which are maintained and solved incrementally during execution by specialized constraint solvers [Wallace, 2002].

The motivation behind CLP is to overcome limitation of pure logic programming when dealing with arithmetic relations, domain restrictions, and combinatorial search problems. In traditional Prolog, such relations are usually solved by using generate-and-test patterns where possible values are generated and then tested against conditions. This approach can be very inefficient, as many of the candidate values generated may violate constraints that are only discarded late into the computation.

In CLP, the basis of a problem is *constraints*. Problems as a whole are represented as a collection of constraints together. Unification is one special case of constraint solving, which means that CLP has superior expressive power [Jaffar and Lassez, 1987]. When a constraint is introduced, it is added to a constraint store, which contains the current set of conditions that must hold. A constraint solver is then responsible for propagating constraints, narrowing variable domains, and detecting inconsistencies for the given constraints as early as possible.

Unlike unification, which assigns concrete values to variables, constraints may leave a variable partially instantiated, for example, a domain variable. The built-in solver for the $=$ constraint is unification, but CLP offers built-in solvers for other kinds of constraints [Wallace, 2002]. This

¹ Available online at: <https://www.swi-prolog.org/>

² Available online at: <https://SICStus.sics.se/index.html>

allows **CLP** programs to delay decisions until sufficient information is available, preventing the use of the previously mentioned inefficient generate-and-test patterns, allowing for a more efficient and declarative exploration of the solution space.

CLP systems are typically made for a constraint domain that determines the types of constraints supported and the algorithms to solve them. **CLP(X)** stands for constraint logic programming over the domain X . Plain Prolog can be described as **CLP(H)** where H stands for *Herbrand* terms. Over this domain, `=/2` and `dif/2` are the most important constraints that express, respectively, equality and disequality of terms. Plain Prolog can thus be regarded as a special case of **CLP** [SWI-Prolog **CLP Documentation**³]. Other common constraint domains include:

- **CLP(FD)**, which supports constraints over *Finite Domains* (typically integers);
- **CLP(R)**, which supports constraints over *Real* ($\mathbb{R}$) numbers;
- **CLP(Q)**, which supports constraints over *Rational* ($\mathbb{Q}$) numbers;
- **CLP(B)**, which supports constraints over *Boolean* variables;

Each domain is associated with a specialized solver designed to efficiently handle the corresponding constraint types. By selecting the appropriate domain for an application, programmers can express complex relationships declaratively while relying on the solver to manage consistency and propagation of those constraints.

These differences between **LP** and **CLP** might seem purely technical with no real benefits, but the key advantage of **CLP** is its ability to prune the search space early before generating possible values, by the use of constraint propagation. The real benefit of **LP** is its ability to correctly capture complex problems, but unfortunately, it has often failed to do so in an efficient way because the natural encoding of a problem in **LP** rarely maps to an efficient and scalable solution [Wallace, 2002]. **CLP** still takes advantage of that benefit by maintaining its declarative structure but implements the solutions in a more efficient and scalable way.

2.2.4 CLP in Prolog

Modern Prolog systems provide extensive support for **CLP** through dedicated libraries that integrate constraint solving into the Prolog execution model. These libraries extend the language with constraint operators and solvers that work alongside unification and backtracking intrinsic to Prolog. Both SWI-Prolog⁴ and SICStus⁵ Prolog have libraries that implement **CLP** for various different constraint domains. Both include libraries for all of the previously defined constraint domains.

1 `:- use_module(library(clpfd)).`

³Available online at: <https://www.swi-prolog.org/pldoc/man?section=clp>

⁴Available online at: <https://www.swi-prolog.org/>

⁵Available online at: <https://SICStus.sics.se/index.html>

```

2 :- use_module(library(clpr)).
3 :- use_module(library(clpq)).
4 :- use_module(library(clpb)).

```

Listing 2.8: SICStus Prolog **CLP** libraries

The libraries allow for the seamless integration of **CLP** into Prolog. For example, in Listing 2.9 there is a Sudoku solver implementation with **CLP** in Prolog. The most important constraint it uses is the `domain/3` predicate given by the `clpfd` library. This predicate receives a list of variables, a minimum value, and a maximum value. It then applies a constraint to all variables in the list, where they have to be between the given range (inclusive). Another extremely important constraint is the `all_distinct/1` predicate. As the name suggests, it receives a list of variables and applies a constraint where all of those variables have to be different from one another.

```

1 :- use_module(library(clpfd)).
2 :- use_module(library(lists)).
3
4 sudoku(Rows) :-
5     length(Rows, 9), maplist(same_length(Rows), Rows),
6     append(Rows, Vs), domain(Vs, 1, 9),
7     maplist(all_distinct, Rows),
8     transpose(Rows, Columns),
9     maplist(all_distinct, Columns),
10    Rows = [As,Bs,Cs,Ds,Es,Fs,Gs,Hs,Is],
11    blocks(As, Bs, Cs),
12    blocks(Ds, Es, Fs),
13    blocks(Gs, Hs, Is).
14
15 blocks([], [], []).
16 blocks([N1,N2,N3|Ns1], [N4,N5,N6|Ns2], [N7,N8,N9|Ns3]) :-
17     all_distinct([N1,N2,N3,N4,N5,N6,N7,N8,N9]),
18     blocks(Ns1, Ns2, Ns3).
19
20 problem(1, [[_,_,_,_,_,_,_,_,_],
21             [_,_,_,_,_,3,_,8,5],
22             [_,_,1,_,2,_,_,_,_],
23             [_,_,_,5,_,7,_,_,_],
24             [_,_,4,_,_,_,1,_,_],
25             [_,9,_,_,_,_,_,_,_],
26             [5,_,_,_,_,_,_,7,3],
27             [_,_,2,_,1,_,_,_,_],
28             [_,_,_,_,4,_,_,_,9]]).

```

Listing 2.9: Sudoku solver with **CLP** in SICStus Prolog

```

1 | ?- problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
2 [9, 8, 7, 6, 5, 4, 3, 2, 1].
3 [2, 4, 6, 1, 7, 3, 9, 8, 5].
4 [3, 5, 1, 9, 2, 8, 7, 4, 6].
5 [1, 2, 8, 5, 3, 7, 6, 9, 4].
6 [6, 3, 4, 8, 9, 2, 1, 5, 7].
7 [7, 9, 5, 4, 6, 1, 8, 3, 2].
8 [5, 1, 9, 2, 8, 6, 4, 7, 3].
9 [4, 7, 2, 3, 1, 9, 5, 6, 8].
10 [8, 6, 3, 7, 4, 5, 2, 1, 9].
11 Rows = [[9,8,7,6,5,4,3,2,1],[2,4,6,1,7,3,9,8,5],[3,5,1,9,2,8,7,4|...],
          [1,2,8,5,3,7,6|...],[6,3,4,8,9,2|...],[7,9,5,4,6|...],[5,1,9,2|...],
          [4,7,2|...],[8,6|...]] ? ;
12 no

```

Listing 2.10: Sudoku solver example query result

As shown in Listing 2.9, with just 2 simple constraints applied to the correct set of variables, one is able to produce a sudoku solver that works flawlessly and gives the correct result (Listing 2.10).

```

1 | ?- problem(1, Rows), sudoku(Rows), maplist(portray_clause, Rows).
2 [_,_,_,_5,_,_,_1].
3 [2,4,6,1,7,3,9,8,5].
4 [_5,1,_2,_,_,_,_].
5 [1,2,_5,_7,_9,4].
6 [_,_4,_9,_1,5,_].
7 [_9,5,4,_1,_,_,_].
8 [5,1,9,_,_4,7,3].
9 [4,_2,_1,9,5,6,8].
10 [_,_,_4,5,2,1,9].
11 Rows = [[_A,_B,_C,_D,5,_E,_F,_G,1],[2,4,6,1,7,3,9,8,5],[_H,5,1,_I,2,_J,_K,_L|...],
          [1,2,_N,5,_O,7,_P|...],[_Q,_R,4,_S,9,_T|...],[_V,9,5,4,_W|...], [5,1,9,_A1
          |...],[4,_D1,2|...],[_F1,_G1|...]],
12 _A in{3}\(6..9),
13 _B in{3}\(6..8),
14 _C in{3}\(7..8),
15 _D in{2}\{6}\(8..9),
16 _E in{2}\{4}\{6}\{8},
17 _F in{3}\(6..8),
18 _G in 2..4,
19 _H in{3}\(6..9),
20 ...
21 _G1 in{3}\(6..8),
22 _H1 in{3}\(7..8),
23 _I1 in{3}\{7} ? ;
24 no

```

Listing 2.11: Sudoku solver without line 11 query result

One of the most important predicates of the `clpfd` library is the `labeling/2` that, given the labeling options and the variables to be labeled, finally tries the possible values in a generate-and-test pattern. The `labeling/2` predicate will systematically return all possible solutions.

If, for example, line 11 from Listing 2.9 (`blocks(As, Bs, Cs),`) was removed, we would get a result like shown in Listing 2.11, where multiple variables would be left unlabeled. Even though any Sudoku without the `all_distinct` restriction on the top boxes still only has one solution, the **CLP** solver isn't able to find it. The fact that it only has one solution is easily provable by the use of Set Theory. If we only consider the first 3 columns of the puzzle, after defining the `all_distinct` for the columns, we have 3 sets of the digits 1 to 9. After defining the `all_distinct` that the 2nd and 3rd boxes (counting down), each of the boxes contains 1 set of the digits 1 to 9. Therefore, of the initial 3 sets of the digits 1 to 9, the last set is in the 1st box. Despite the fact that the Sudoku is still solvable, the solver can't find the solution with the given constraints. In scenarios like this, one must either add constraints (if possible) or use the `labeling/2` predicate to systematically iterate through the options and check for a solution.

If at the end of the `sudoku/1` predicate, the line `labeling([], Vs).` was added, the solver would try possible values to check for a solution. The program resulting from this change and the change mentioned above is a program that still gives the same result as the first program (Listing 2.10), but with the use of generate-and-test strategies, which make it more inefficient.

2.3 Summary

This chapter introduced the foundational concepts required to support the remainder of this dissertation. It introduced core software testing principles and contrasted Unit Testing (**UT**) with Property-Based Testing (**PBT**), highlighting the latter's focus on properties and automated input generation. The limitations of purely random testing and the motivation for targeted approaches were also discussed. Following that came an introduction to Logic Programming (**LP**), followed by Constraint Logic Programming (**CLP**) as an extension that integrates constraint solving into logical inference. Also introduced Prolog as a programming language from the **LP** paradigm, and its capabilities when it comes to **CLP**. These concepts establish the theoretical basis for using **CLP** to support automated and targeted test data generation in Prolog in the chapters that follow.

Chapter 3

Data Generation with CLP

This chapter presents the state-of-the-art for data generation techniques that take advantage of **CLP**, especially focused on efficient random structurally complex test data generation with **CLP**. For a complete and recent state-of-the-art analysis for input generation specifically for **PBT**, see [de Sá, 2025].

Data Generation is the backbone of input generation, which is a crucial component of **PBT**, as it is the first step in the **PBT** pipeline. Improving such a process with the use of **CLP** might be interesting. To gather relevant documents and establish the state-of-the-art in existing algorithms regarding data generation with **CLP**, a search was conducted in three key research databases: the Association for Computing Machinery Digital Library (ACM DL)¹, IEEE Xplore² from the Institute of Electrical and Electronics Engineers (IEEE), and Scopus³. 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 contained algorithms or strategies that directly use **CLP** concepts for the data generation process, focused on structured data, and without the use of White-box testing strategies to define constraints. The queries focused on article abstracts because they typically summarize the paper's main contributions, techniques, and concepts. There were 2 queries: after the first, some terms, such as "test case generation", weren't being captured, which led to the creation of a second query that focused on those cases. Also, IEEE Xplore's advanced search has a limit of 10 wildcards, which would make the query that's equivalent to both queries invalid, therefore requiring 2 queries anyway. ACM DL's search doesn't allow for wildcards inside search terms with quotation marks ("), so the most common search terms were used instead of using wildcards.

The exact queries used are presented in Table A.1 in Appendix A but the equivalent general query is the following:

```
("Abstract": "data-driven test*")
```

¹ Available online at: <https://dl.acm.org>

² Available online at: <https://ieeexplore.ieee.org>

³ Available online at: <https://www.scopus.com>

```

OR "Abstract": "data" NEAR/1 "generat*"
OR "Abstract": "data" NEAR/1 "creat*"
OR "Abstract": "input" NEAR/1 "generat*"
OR "Abstract": "input" NEAR/1 "creat*"
OR "Abstract": "test*" NEAR/1 "generat*"
OR "Abstract": "test* case* generat*"
OR "Abstract": "test*" NEAR/1 "creat*"
OR "Abstract": "test* case* creat*")
AND ("Abstract": "constraint" NEAR/1 "program*"
OR "Abstract": "constraint-based"
OR "Abstract": "constraint logic program*")

```

The quantitative results of the search conducted in January 2026, after merging both queries and eliminating duplicate documents (within each database, not across databases), are presented in Table 3.1. From these queries resulted 57 interesting unique papers, to which were added 3 other papers given by the supervisor teachers or found by other means, for a total of 60 papers. This "interesting" paper category represents a pre-selection made while sorting through all the documents. This category contains any paper that mentions data generation with **CLP** in its title or abstract.

Research Database	Total Results	Interesting Results	Relevant Results
ACM Digital Library	116	26	2
IEEE Xplore	77	19	1
Scopus	302	54	4
Other Means	3	3	1
TOTAL (Unique)	337	57	5

Table 3.1: Quantitative results of queries on Data Generation with CLP

Of the 60 initially interesting papers found, only 5 are relevant to the problem at hand, that is, efficient structurally complex test data generation with **CLP** for a Black-box testing technique. Most of the non-relevant papers use **CLP**, or other constraint techniques, to apply constraints derived from the source code, either by the use of Symbolic Execution, Structural Testing, Mutation Analysis, or a combination of them, which makes them White-box testing techniques. The 5 relevant papers can be divided into two categories: Bounded-Exhaustive Testing (**BET**) (4 papers) and a novel Constraint-Based Random Testing technique (1 paper). From these results, there is clearly a gap in development on Constraint-Based Random Data Generation with only 1 relevant paper, and also a gap in Targeted Input Generation with **CLP** strategies. The 5 relevant papers will be discussed in the remainder of this chapter.

3.1 Bounded-Exhaustive Testing

Bounded-Exhaustive Testing (**BET**) is an automated testing methodology based on the observation that software defects often manifest in small-scale instances of complex data structures [Sullivan

et al., 2004]. The primary goal of **BET** is to generate and test all possible input instances that satisfy a given invariant within a specified size bound. This approach is particularly important for modern software that manipulates structurally complex data, such as balanced trees or graphs, where manual test case generation is error-prone and expensive [Senni and Fioravanti, 2012].

Korat is a tool for **BET** of structurally complex data structures for test inputs in Java programs [Milicevic et al., 2007]. Korat receives an *imperative predicate* that expresses a structural invariant in Java code and a *finitization* that bounds the size of the structures that are inputs to the predicate. This predicate should return `true` if the data structure satisfies its representation invariant. The solver then systematically searches the bounded structure space, effectively pruning large portions of that space, and outputs all nonisomorphic structures for which the predicate returns `true`. This is the reason it's called *Bounded-Exhaustive Testing* because it gives an exhaustive test suite in the given bounds.

Korat works by pruning when the predicate returns `false` and backtracking to the last fields accessed by the predicate. For example, if a sorted linked list data structure has its first two elements unsorted, Korat will prune all possible descendants of this possibility because the last accessed element was the second. This makes **BET** possible because an exhaustive search of the possibilities would quickly become unfeasible. This idea was expanded by Aguirre et al. in 2011 [Aguirre et al., 2011] by also pruning when the predicate returns `true` by creating a function that returns the equivalence class for a given input, and monitoring the accessed fields, it then prunes all possible descendants of the last accessed fields. This idea, however, doesn't take advantage of Constraint Programming (**CP**) techniques, so it's not going to be discussed further.

The first use of **CLP** for **BET** came from a paper from 2012 by Senni and Fioravanti [2012], later polished and published in 2015 with Proietti [Fioravanti et al., 2015]. Their approach for developing test case generators takes advantage of the CLP(FD) programming paradigm, sometimes referred to as constrain-and-generate [Marriott and Stuckey, 1998]. It follows a symbolic approach, and it's structured mainly in 2 phases.

In the first phase, a so-called *shape* of the desired data structure is constructed, capturing basic structural properties that make the structure correct (such as being a binary tree or a list), leaving parts of the data structure undefined by using variables. Then constraints are added to those variables derived from invariants that define the data structure, and a so-called *constrained shape* is built. It's at this point that the constraints are checked against each other by the built-in (incomplete) solver, allowing for inconsistent shapes to be rejected early (*constrain* phase).

In the second phase, a complete solver is invoked to find the final shapes that can effectively be instantiated to a concrete data structure, and from there, all the correct instantiations are computed, which are the final set of test cases for **BET** (*generate* phase).

The authors provide a template based on this constrain-and-generate paradigm for developing (filter-based) test case generators. The template is divided into three parts. The **Preamble** contains the definition of the list of previously mentioned variables and their domains. The **Symbolic Definition** contains a call to a generator which defines the data structure shapes (e.g., list, tree, graph) and a sequence of calls to filters defining the invariants in terms of constraints among the

variables. The **Instantiation** contains the calls to the complete solver to instantiate the variables. The template is defined in Listing 3.1.

```

1 tc(T,P1 ,...,Ph) :-
2     % Preamble                                (constrain)
3     init(P1 ,...,Ph,V1 ,...,Vk), % definition of domains
4                                     % of the FD -variables
5     % Symbolic Definition                    (constrain)
6     g(T,P1 ,...,Ph,V1 ,...,Vk), % shape
7     invl(T,P1 ,...,Ph), ..., % invariants
8     invn(T,P1 ,...,Ph),
9     % Instantiation                          (generate)
10    labeling(V1) ,...,
11    labeling(Vk) .

```

Listing 3.1: Template for developing test case generators [Fioravanti et al., 2015]

In Listing 3.1, the predicate `tc` is the generator that returns the data structure on the first argument `T`; it receives any number of parameters `Pn`. The predicate `init` is responsible for instantiating the variables with the given parameters; it receives any number of parameters `Pn` and returns any number of variables `Vn`. The predicate `g` defines the shape of the data structure; it returns the data structure on the first argument `T`, receives any number of parameters `Pn`, and receives any number of Variables `Vn`. The predicates `invl`, ..., `invn` apply the constraints derived from invariants of the data structure; they receive the shape of the data structure `T` and receive any number of parameters `Vn`. The predicate `labeling` receives the variables `Vn` and instantiates them. This template allows for the creation of a data generator with **CLP** for any arbitrary data structure.

3.1.1 Example of CLP generator for Red-Black Trees

In this section, a concrete example is discussed to showcase their approach. This example is taken directly from [Fioravanti et al., 2015].

A Red-Black tree is a binary search tree where each node has two labels: a *color*, either red or black, and an integer, called *key* (for the purpose of test generation, node values are abstracted away in the definition of the data structure). Therefore, it satisfies the following type constraints in Listing 3.2, where 0 and 1 denote *red* and *black*, respectively, and `e` denotes the empty tree.

```

1 Color ::= 0 | 1
2 Key   ::= ... | -1 | 0 | 1 | ...
3 Tree  ::= e | Color x Key x Tree x Tree

```

Listing 3.2: Red Black Tree Definition

A Red-Black tree must also satisfy the following three invariants:

- (I₁) every path from the root to a leaf has the same number of black nodes
- (I₂) no red node has a red child
- (I₃) for every node n , all the nodes in the left (respectively, right) subtree of n , if any, have keys which are smaller (respectively, bigger) than the key labeling n .

The **CLP** specification of a Red-Black tree generator is parameterized by the minimum and maximum tree size (number of nodes), and by the maximum key value. In Listing 3.3, `rbtree` defines the test case generator.

```

1 rbtree(T, MinSize, MaxSize, NumKeys) :-
2   % Preamble
3   MinSize #=< S, S #=< MaxSize,
4   varlist(S, Keys), varlist(S, Colors), Max #= NumKeys - 1,
5   domain(Keys, 0, Max), domain(Colors, 0, 1),
6   % Symbolic Definition
7   lbt(T, S, Keys, [], Colors, []),          % shape
8   pi(T, _), ci(T), ordered(T, 0, NumKeys), % invariants
9   % Instantiation
10  labeling(Keys), labeling(Colors).
```

Listing 3.3: Red Black Tree Definition

Given the non-negative input integers `minSize`, `maxSize`, and `numKeys`, the set of answers given by `rbtree(T, minSize, maxSize, numKeys)` contains all the Red-Black trees of size between `minSize` and `maxSize`, with keys ranging from 0 to `numKeys-1`. The predicate `varlist` is used for constructing a list of fresh variables.

The first line of the **Preamble** fixes the domain of the variable `S` denoting the tree size. Then, two lists of variables are defined, `Keys` and `Colors`, with the corresponding domains, 0 to `NumKeys-1` and 0 and 1, respectively. These variables are placed along the tree structure in the **Symbolic Definition** part of the predicate `lbt`, which defines (2-)labeled binary trees by structural induction (Listing 3.4).

```

1 lbt(e, S, Ks, Ks, Cs, Cs) :- S #= 0.
2 lbt(t(C, K, L, R), S, [K1 | Ks], NKs, [C1 | Cs], NCs) :-
3   S #>= 1, SL #>= 0, SR #>= 0, S #= SL + SR + 1, C #= C1, K #= K1,
4   lbt(L, SL, Ks, TKs, Cs, TCs), lbt(R, SR, TKs, NKs, TCs, NCs).
```

Listing 3.4: Red Black Tree Shape Definition

In Listing 3.4, the first argument is either the constant `e` denoting the empty tree or a term `t(C, K, L, R)` denoting a non-empty tree with left subtree `L`, right subtree `R`, and whose root

node is labeled with color C and key K . The second argument is the size of the tree (number of nodes), which, in line 3, is at least 1, and it is split into a pair of non-negative integers SL and SR denoting the size of the left and right subtrees, respectively, such that $S=SL+SR+1$. The left and right subtrees are then constructed recursively. The remaining two arguments contain the key and color variables that are placed in each node.

```

1 pi(e, C) :- C #= 0.
2 pi(t(C, _, L, R), D) :- ND #>= 0, D #= ND + C, pi(L, ND), pi(R, ND).

```

Listing 3.5: Red Black Tree Path Invariant (I_1)

In Listing 3.5, the predicate `pi` (for path invariant) encodes the invariant (I_1). The semantics of `pi` is the following: for a given tree t , `pi(t, d)` holds if d is the number of black nodes on every root-to-leaf path in t . We say that d is the value of the *black-nodes counter* of t . If a tree is empty, then its black-nodes counter is 0, otherwise, the black-nodes counter is computed by adding the 'color' of the root (i.e., 0 if red, 1 if black). to the black-nodes counter of (both) its subtrees (that must have the same value).

```

1 ci(e).
2 ci(t(C, _, L, R)) :- root_col(L, C), root_col(R, C), ci(L), ci(R).
3
4 root_col(e, _).
5 root_col(t(C, _, _, _), D) :- C + D #> 0.

```

Listing 3.6: Red Black Tree Color Invariant (I_2)

In Listing 3.6, the predicate `ci` (for color invariant) encodes the invariant (I_2). In line 2, the color invariant is enforced by testing the color of the roots of the left and right subtrees. For each branch, the constraint $C+D\#>0$ enforces either the father or the child to be black (i.e., equal to 1).

```

1 ordered(e, _, _).
2 ordered(t(_, N, L, R), Min, Max) :-
3     Min #=< N, N #< Max, M #= N + 1,
4     ordered(L, Min, N), ordered(R, M, Max).

```

Listing 3.7: Red Black Tree Ordered Invariant (I_3)

Finally, in Listing 3.7, the predicate `ordered` defines the invariant (I_3). In the second clause, it forces the left subtree L to be between `Min` and the key of the root node N (i.e. less than N), and forces the right subtree R to be between the key of the root node N and `Max` (i.e. greater than N), therefore making it ordered.

This example shows how straightforward it is to implement the template just by defining the data structure’s shape and its invariants.

3.2 Constraint-Based Random Testing

The generation of uniform and efficient solutions to Constraint Satisfaction Problems (CSPs) is a fundamental part of Property-Based Testing (PBT) because the first step of PBT requires the generation of input data that often has constraints that it needs to abide by. Traditional frameworks, such as QuickCheck [Claessen and Hughes, 2000], circumvent this problem by relying on rejection sampling, which consists of generating first and checking for constraints after. This method is simple but computationally expensive when the solution spaces are sparse.

Recent advancements, proposed by Ziat and Pépin [2025], introduce the first use of Constraint Programming for Random Testing. They describe a *good* generator as one that has three qualities: (1) **correct**, respects constraints they are subject to; (2) **uniform**, explores the solution space uniformly (i.e., without bias); (3) **efficient**, minimizes testing overhead. The main theoretical contributions of this paper are described in the remainder of this section.

3.2.1 Rejection Sampling

The aforementioned rejection sampling, used by frameworks like QuickCheck, succeeds in meeting the first 2 criteria for a good generator, but it fails to meet the third. This is because rejection sampling can lead to significant overhead, in particular when a large portion of the generated values are rejected. A basic pseudo-code implementation is presented for clarity in Algorithm 1. Algorithm 1 takes as input a set of variables V , their associated range of values D , and a set of constraints C and returns a valid candidate.

Algorithm 1 Rejection sampling procedure

```

1: function SAMPLE( $V, D, C$ )
2:    $candidate \leftarrow spawn(V, D)$  ▷ draw at random in ( $V, D$ )
3:   if sat( $candidate, C$ ) then
4:     return  $candidate$ 
5:   else
6:     return sample( $V, D, C$ )

```

3.2.2 Constraint-Based Generators

The authors of [Ziat and Pépin, 2025] propose a constraint-based approach for random sampling that relies on an abstract solving method. Algorithm 2 constructs a cover of the solution space using abstract elements (e.g., boxes, octagons, polyhedra, etc.). This cover consists of two sets: inner elements (I) and outer elements (O). The set I under-approximates the solution set, while $I \cup O$ over-approximates it. The algorithm starts by initializing an abstract element and inserting it into O . It then iterates through the following steps: an element from O is selected, filtered, and, if

it satisfies the constraints, added to I . If it does not, it is split into sub-elements that are reinserted into O . As presented, this algorithm may not terminate, but termination criteria can be added, such as limiting the size of the elements considered or the depth of the solving tree.

Algorithm 2 Abstract solving method

```

1: function SOLVE( $X, D, C$ )                                ▷  $X$ : variables,  $D$ : domains,  $C$ : constraints
2:    $I \leftarrow \emptyset$                                        ▷ inner solutions
3:    $O \leftarrow \{\text{init}(D)\}$                                 ▷ outer solutions
4:   while  $O \neq \emptyset$  do
5:      $e \leftarrow \text{select}(O)$ 
6:      $e' \leftarrow \text{filter}(e, C)$ 
7:     if  $e' \neq \perp$  then
8:       if  $\text{solution}(e', C)$  then
9:          $I \leftarrow I \cup \{e'\}$ 
10:      else
11:         $O \leftarrow O \cup \text{split}(e')$ 
12:   return  $I, O$ 

```

3.2.2.1 Constraint-Based Sampling

After building the abstract elements divided into inner and outer elements, as described in Section 3.2.2, the authors propose an algorithm for uniform sampling under constraints, as shown in Algorithm 3. It repeatedly selects an element e from either the inner or outer sets using a `select` function. Here, `select` performs a weighted choice based on the volume of the elements, with the largest elements having the best chance of being chosen. The algorithm generates a candidate value i within e . If e belongs to the inner set or i satisfies the constraints C , the algorithm returns i ; it repeats the same steps until it finds a solution.

Algorithm 3 Random Sampling for Covers

```

1: function GENERATE( $inner, outer, C$ )
2:   while true do
3:      $e \leftarrow \text{select}(inner, outer)$ 
4:      $i \leftarrow \text{spawn}(e)$ 
5:     if  $e \in inner \vee \text{sat}(i, C)$  then
6:       return  $i$ 

```

The Algorithm 3 ensures uniform sampling under three conditions: the elements in $I \cup O$ must not overlap, points, within each element e must be sampled uniformly $P(i|e) = \frac{1}{v(e)}$, and the probability of selecting an element e must be proportional to its volume $P(e) = \frac{v(e)}{\sum_{e' \in I \cup O} v(e')}$. These conditions ensure that the sampling process is uniform across the domain, giving all points in the union of elements an equal probability of being chosen.

3.2.2.2 Efficient selection of an abstract element

To implement the `select` function, a data structure for storing a collection of abstract elements that supports efficient sampling of elements with probability proportional to their volume, and an efficient way to replace an abstract element with a collection of smaller elements, is required. For this purpose, the authors propose the use of a binary tree that stores the abstract elements at the leaves and the other nodes save the sum of the volumes of their children. Using such a data structure, drawing an abstract element proportional to its volume corresponds to drawing a uniform real variable between 0 and the total volume of the tree, and to recursively descend the tree, choosing between the left and right child based on their volumes, as shown in Algorithm 4.

Algorithm 4 Random generation of an abstract element using the tree data structure

```

1: function RANDOMTREeselect( $T$ )
2:    $r \leftarrow \text{UNIF}(0, \text{volume}(T))$ 
3:   return TREeselect( $r, T$ )

```

Require: $0 \leq r < \text{volume}(T)$

```

1: function TREeselect( $r, T$ )
2:   if  $T$  is a leaf then
3:     return the abstract element stored in  $T$ 
4:    $(T_L, T_R) \leftarrow$  the children of  $T$ 
5:   if  $r < \text{volume}(T_L)$  then
6:     return TREeselect( $r, T_L$ )
7:   else
8:     return TREeselect( $r - \text{volume}(T_L), T_R$ )

```

In Algorithm 4, the cost of selecting an element is proportional to the length of the path from the root to the selected leaf. Given the volume of every abstract element (and thus their probability of being drawn), there is an optimal way to arrange these elements in the tree to minimize the expected cost of the generation. The authors recognize that the optimal way to organise the tree for this purpose would be to use a Huffman tree [Lampropoulos et al., 2018], but those properties are hard to maintain efficiently when we update the collection of abstract elements, which is needed. To mitigate this before doing any sampling, a first solving pass is done until a certain depth, after which a Huffman tree T is constructed based on the resulting volumes, then during the iterative sampling process, every time a split of an abstract element e into smaller elements $e_1, \dots, e_n$ is needed a Huffman tree T' for $e_1, \dots, e_n$ is constructed and that tree replaces the abstract element. The authors expect that this strategy, despite not maintaining the optimal Huffman Tree, maintains some balance in the tree.

3.2.2.3 Incremental and Adaptive Generators

Constraint-solving mechanisms are sometimes computationally expensive to execute, so the authors propose the use of an iterative solver instead of attempting to solve the entire problem at once. This works by first prioritizing certain parts of the problem, identified on the fly, whose filtering can greatly improve the performance of random generation, then, secondly, they attempt

to amortize the cost of the resolution steps (filtering and exploration) during generation. For example, every time a generator is called, a resolution step can be performed; this way, the more a generator is used, the more it improves. To achieve this, the generator saves an internal solving state that evolves with each step and retains information from previous resolution steps, allowing the generator to improve progressively rather than starting from scratch every time.

For each element in the cover, the number of times it was selected and the number of times it successfully produced a sample are recorded. When the algorithm fails to produce a valid solution within a selected element, we must decide if it is necessary to split it or not. The authors propose basing our choice on its acceptance rate. If it falls below the global acceptance rate, we proceed to refine that element, meaning the element is split and replaced by its sub-elements, otherwise, it's left unchanged. This dynamic adjustment of the refinement process can lead to faster convergence and avoids over-splitting, which can deteriorate the element selection procedure. This is shown by Algorithm 5.

Algorithm 5 Sampling-Guided solving

```

1: function GENERATE(inner, outer, C)
2:   while true do
3:      $e \leftarrow \text{select}(\textit{inner}, \textit{outer})$ 
4:      $i \leftarrow \text{spawn}(e)$ 
5:     if  $e \in \textit{inner} \vee \text{sat}(i, C)$  then
6:       return  $i$ 
7:     else if  $\text{rate}(e) < \text{global}(\textit{inner}, \textit{outer})$  then
8:       refine  $e$ 

```

3.2.3 Graph Representation of a CSP for Random Sampling

The authors of [Ziat and Pépin, 2025] propose the use of a graph representation of the CSP to successfully identify independent sub-problems that can be generated separately and then have the results joined.

The constraint hypergraph of a constraint satisfaction problem is a hypergraph in which the vertices correspond to the variables, and the hyperedges correspond to the constraints. Hypergraph representations can be integrated with various constraint-solving algorithms, such as backtracking, constraint propagation, and local search. An interesting idea for the use of the hypergraph representation is the detection of Connected Components (CCs). A partition of the constraint hypergraph into CCs corresponds to a partition of the CSP into independent sub-problems, effectively decoupling the sampling for each component. This means that, if the constraint graph consists of two (or more) CCs, it is then possible to generate solutions independently in the different components and then combine the results. Intuitively, leveraging this independence should speed up the random generation process. The authors also give a heuristic argument for this speed-up.

In practice, at initialization, the CSP is divided into (disjoint) CCs, and a sampling cache is associated with each one. This cache stores the partial samples generated for each component, allowing the algorithm to reuse previously computed samples and avoid redundant computations.

This idea can also be beneficial by using the cache to identify and prioritize the solver's efforts on the sub-problems that exhibit higher rejection rates. Empty caches indicate difficulty in finding valid samples, encouraging the solver to select a variable to split in these components. By adapting to the performance of individual components, the solver converges towards a locally optimal configuration.

3.2.3.1 The *bridge-first* split

The authors also point out that the iterative process of splitting and filtering gradually ensures that certain constraints are locally satisfied, allowing them to be removed from the graph completely. Whenever a constraint is removed, the solver checks whether its removal disconnects the graph. When this is the case, the graph is split into **CCs**, enabling their independent handling.

To achieve this, the authors developed a propagation scheme and an exploration heuristic that help reduce the connectivity of the constraint graph. For each edge, information on whether it is a bridge or not is saved. A bridge is an edge whose removal increases the number of **CCs** in the graph. Such bridges are identified using Tarjan's algorithm [Tarjan, 1974]. Tracking which edges are bridges is useful for guiding exploration, since variables that are part of a bridge are particularly interesting for splitting. When a constraint is removed, if it was a bridge, the number of **CCs** are, by definition, going to increase, and as such **CCs** are recomputed. Otherwise, the removal is checked to determine whether it introduced new bridges, without recalculating the **CCs**. If a bridge is detected, the solver strives to eliminate it because its removal would disconnect the graph and thus improve the sampler's performance. This is done by prioritizing the variables in the bridge's support during splitting steps.

3.3 Summary

This chapter reviews the state-of-the-art in structured data generation using Constraint Logic Programming (**CLP**), with particular emphasis on approaches suitable for Black-box testing and **PBT**. From an initial pool of 60 interesting papers, only 5 directly address the problem of efficient generation of structurally complex test data using **CLP** without relying on White-box information. Two main methodologies were identified: Bounded-Exhaustive Testing (**BET**), which uses a "constrain-and-generate" paradigm to systematically explore all valid data structures with fixed bounds, and Constraint-Based Random Testing, which improves upon rejection sampling by using abstract solvers and iterative solvers to explore sparse solution spaces more efficiently.

Bounded-Exhaustive Testing (**BET**) uses a systematic and exhaustive strategy within predefined bounds. The "constrain-and-generate" paradigm, shown by the **CLP**(FD)-based approach of Fioravanti et al. [2015], separates the structural construction from invariant enforcement and the instantiation of an actual data structure. This separation allows for early pruning of invalid data structures before instantiating them through the use of constraint propagation.

Constraint-Based Random Testing focuses on uniform and efficient sampling of constrained solution spaces. By replacing rejection sampling with abstract constraint solving, this approach

divides the solution space into sections where all of the points satisfy the conditions (inner) and regions on the edge that contain points that satisfy the CSP and ones that don't satisfy the CSP (outer). These regions are then sampled by using weighted sampling based on their volumes. This methodology allows for the use of constraint-solving techniques to efficiently reduce the possibilities, but also allows for random sampling of the solution space in an efficient way.

Overall, when it comes to the use of these techniques for PBT, Constraint-Based Random Testing is much more interesting because PBT uses random or partially random inputs for its tests, but the approach shown for BET states a systematic way to define constraints for any data structure, so together they are interesting for PBT.

Chapter 4

Methodology

To answer the proposed research questions, this dissertation aims to continue the development of the Property-Based Testing (PBT) framework for Prolog [de Sá, 2025] by identifying the most promising algorithms for data generation with CLP as a solution to the efficient structured input generation problem.

4.1 Proposed Solution and Architecture

The continuation of the implementation of a Property-Based Testing (PBT) library created especially for SICStus Prolog is the basis of the suggested approach, focused on the development of a generation engine that replaces standard random generators with constraint-based generators.

Given that some of the most interesting algorithms found during the theoretical research are very recent, implementations are not plentiful. Therefore, this dissertation aims to implement them in the Prolog language and use them for the continued development of the aforementioned framework.

4.2 Planning and Work Organization

The objective of this work is to enhance input generation for Property-Based Testing in Prolog by integrating Constraint Logic Programming (CLP) and maybe targeted search techniques, so this work was divided into a theoretical analysis stage and a development stage. The theoretical analysis was already conducted, so the development stage remains.

4.2.1 Theoretical Analysis

To answer the first research question **RQ1** (section 1.4), a theoretical review of the algorithms used for data generation with CLP was conducted, focusing on structured data in a Black-box environment. The goal was to find the most suitable algorithms for input generation using CLP for the development of the PBT framework.

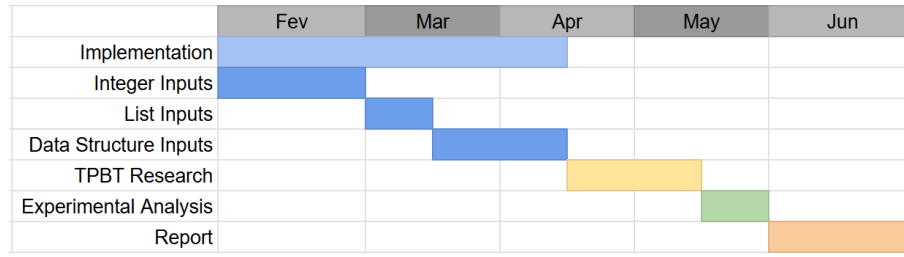


Figure 4.1: Gantt Chart for the Work Plan

4.2.2 Development

The work plan for the development phase consists of 2 main stages, which are the implementation of the techniques in the Prolog framework and then the experimental analysis of the resulting framework by comparing it to the previous version with only random input generation. The implementation stage is divided into 3 main parts: first, implementing the techniques for integer inputs, then for lists, and for data structures, and after all of those. The initial implementation will be with only integer inputs and will be expanded on. This allows for an initial prototype that integrates the **CLP** features into the current framework. If the work goes according to plan and there is enough time, a **TPBT** Research period, where an attempt to design and implement **TPBT** strategies, will ensue. The Gantt Chart for the work plan is displayed in Figure 4.1.

4.3 Experimental Process and Evaluation Design

To validate the effectiveness of the proposed approach, a comparative experimental study will be conducted. This study will compare the new **PBT** framework with the old version, as well as compare the new framework with a Unit Testing (**UT**) benchmark. They will be compared on execution efficiency (execution time) as well as their fault detection capabilities.

4.4 Expected Results and Relevance

It is expected that the integration of **CLP** will provide a substantial performance boost when testing predicates that require highly structured data, by narrowing the search space through constraint propagation. The relevance of this work lies in its potential to bring modern software testing to the Prolog ecosystem, making it more viable for safety-critical applications where data integrity and logical correctness are extremely important.

4.5 Risk Analysis

The proposed solution could face some problems when the time comes to actually implement it. The main risks are:

- **R1:** Friction in the integration process with the existing PBT tool, needing changes or refactoring of the original code;
- **R2:** **CLP** won't be expressive enough to model complex data structures or knowledge bases that a human is capable of crafting;
- **R3:** **CLP** won't be efficient enough for complex problems, causing the testing tool to become cumbersome to use;
- **R4:** The user of the testing tool will need to write generators for every possible data structure.

Table 4.1 displays the risks, along with their respective likelihood of occurrence and impact on the development process if they come to fruition. It also proposes ways to mitigate these risks.

Risk	Likelihood	Impact	Mitigation
R1	Medium	High	Develop a prototype early that integrates correctly with the existing framework, and iterate continuously on it with continuous testing.
R2	Medium	High	Extend the CLP model with domain-specific custom constraints that increase the expressiveness of the solver.
R3	Medium	High	Optimize constraint models and perform performance benchmarking throughout development.
R4	High	Medium	Provide reusable generator libraries, default generators for common data structures, and detailed documentation and examples. In addition to that, create a way for the user to create a generator from scratch for any arbitrary data structure.

Table 4.1: Risks and mitigation strategies

4.6 Summary

This chapter outlines the methodology of this dissertation, focused on enhancing a Prolog Property-Based Testing (**PBT**) framework by replacing standard random generators with Constraint Logic Programming (**CLP**) engines. The approach transitions from a "theoretical analysis" of state-of-the-art **CLP** algorithms to a "development stage" focused on implementing these algorithms within SICStus Prolog. The development will start with a basic prototype capable only of generating integer inputs and will be improved, increasing its range, throughout the development process. To validate the solution, an experimental design is proposed to benchmark the new framework against its predecessor and traditional Unit Testing, measuring improvements in execution efficiency and fault detection. The work aims to provide a high-performance testing tool suitable for safety-critical Prolog applications.

Chapter 5

Development of Database Generation for a PBT Library with CLP

This chapter describes the development process for the integration of **CLP** for database generation in a **PBT** Library. It first explains the main architecture of the final implementation and then explains each new component in detail. The development environment used was SICStus Prolog, mainly because the previous work used it as well.

5.1 Main Architecture

The main architecture of the library is very similar to the previously developed library [de Sá, 2025]; the main difference is the division between constraining and generating arguments, a *constrain-and-generate* approach [Marriott and Stuckey, 1998] for database generation. In Figure 5.1, we have the updated architecture of the developed **PBT** Library based on the architecture shown in the previous work on the library. The Input Constraints and Database Constraints modules are responsible for constraining the arguments to the property and the databases, respectively. The Generator module, as the name suggests, is responsible for generating the arguments as well as the databases. The Shrinker module is responsible for shrinking the arguments if a property fails. The Meta Properties module includes predicates such as `solutions_in_range(Min, Max, Generator, Template)`, which can be used to define properties that count solutions, for example, which are not relevant for the main topic of this thesis, so it will not be mentioned again.

Comparing the new architecture with the one present in the previous work [de Sá, 2025], apart from the division of the constraint and generation processes, the other difference in the architecture is the replacement of the "Knowledge Base Manager" module with the "Database Constraints" module, which serves the same purpose of ultimately generating knowledge bases.

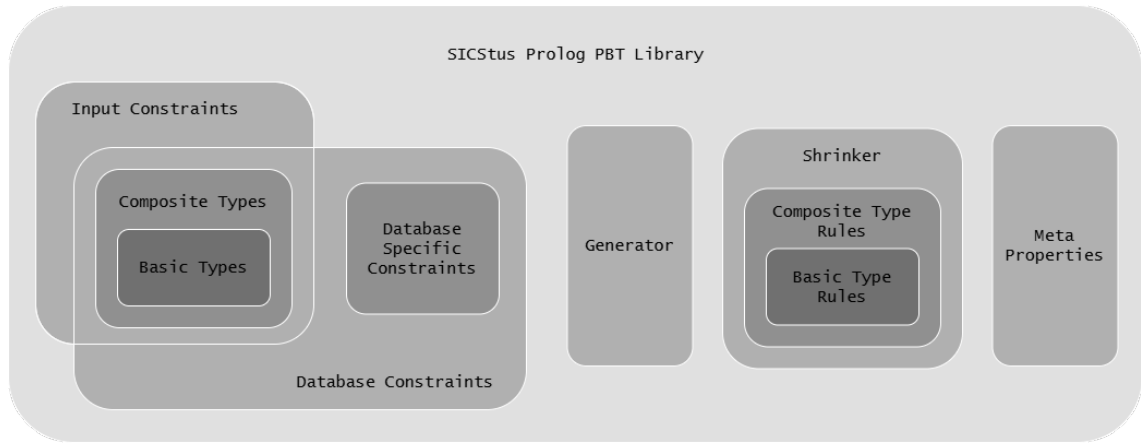


Figure 5.1: Updated Architecture of the developed PBT Library

To run a property test, it starts by applying the constraints to the arguments and the databases. After the constraints are applied, each test is run. Inside a single test run, the Generator is called to instantiate both the arguments to the property as well as the databases, and the property is called. If the property fails, it passes the arguments that caused the property to fail to the Shrinker, which is then printed to the user. If it succeeds, then the instantiations are backtracked over to restore the constraints, and the next test is run. A flow graph of this process is presented in Figure 5.2.

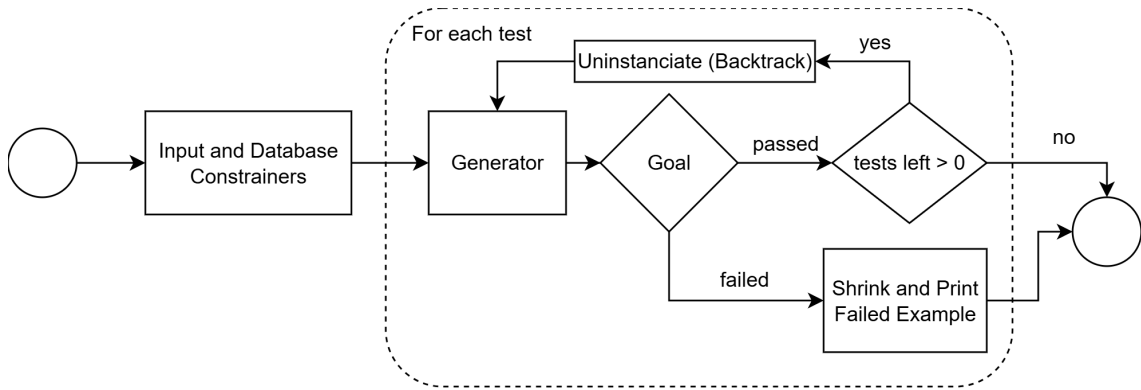


Figure 5.2: Flow Graph of the Main Testing Workflow

5.2 CLP for Basic and Composite Data Types

The previously developed library [de Sá, 2025] already supported generating basic and composite data types, where basic types are primitive types and primitive structures, such as integers and lists, and composite types are built from one or more basic types.

The predicates `arbitrary/2` and `composite/3` previously developed were generators, which means that after being called, they would return a random value from the given type. To implement the *constrain-and-generate* approach, instead of being generators, they were transformed into constrainers. In addition to these 2 predicates, a third `arbitrary/3` was added that allows for the constraining of a variable that depends on other variables.

5.2.1 Basic Types

The `arbitrary` predicates maintained their name from the previous iteration of this library, despite them not generating an arbitrary value but instead constraining a variable to the given type. The difference between the predicates `arbitrary/2` and `arbitrary/3` is that `arbitrary/2` receives a type and returns a domain variable of that type, the value and the domain variable that defines it are the same, but `arbitrary/3` receives a type and returns a value variable and a list of domain variables that, once defined, define the value variable. Listing 5.1 contains examples of integer, boolean, and list constraints.

```

1 arbitrary(integer(A, B), X) :-
2     X in A..B.
3 arbitrary(integer, X) :-
4     X in -2147483648..2147483647.
5 arbitrary(positive, X) :-
6     X in 1..2147483647.
7 arbitrary(negative, X) :-
8     X in -2147483648..(-1).
9
10 arbitrary(id, X) :-
11     arbitrary(positive, X).
12
13 arbitrary(boolean, X, [Y]) :-
14     Y in 0..1,
15     when(ground(Y), nth0(Y, [false, true], X)).
16
17 arbitrary(list(T, Size), X, Vars) :-
18     length(X, Size),
19     ( clause(arbitrary(T, _), _)
20     -> maplist(arbitrary(T), X), Vars = X
21     ; maplist(arbitrary(T), X, LVars), append(LVars, Vars)).

```

Listing 5.1: Example of arbitrary constraints

Other arbitrary constraints that are very useful for database generation are the `oneof` and the `listof`. The `oneof` is used to get one random value from a list of values or a predicate. The `listof` predicate is used to get a random list of unique values from a set of possible values. The `oneof` and `listof` arbitraries are shown in Listing 5.2 and Listing 5.3, respectively.

```

1 arbitrary(oneof(L), X, [Idx]) :-
2     is_list(L), !,
3     length(L, N), ground(N),
4     Idx in 1..N,
5     when(ground(Idx), nth1(Idx, L, X)).
6

```

```

7 arbitrary(oneof(PredName/Arity), X, Vars) :- !,
8   arbitrary(oneof(PredName/Arity, []), X, Vars).
9
10 arbitrary(oneof(PredName), X, Vars) :-
11   arbitrary(oneof(PredName/1), X, Vars).
12
13 arbitrary(oneof(PredName/Arity, Args), X, [Idx]) :-
14   length(Args, ArgsLength),
15   ArityM1 is Arity - 1 - ArgsLength,
16   ArityM1 >= 0,
17   length(Rest, ArityM1),
18   append(Args, Rest, FinalArgs),
19   Pred =.. [PredName, Value | FinalArgs],
20   findall(Value, Pred, Values1),
21   length(Values1, N), Idx in 1..N,
22   when(ground(Idx), (
23     findall(Value, Pred, Values2),
24     nth1(Idx, Values2, X))).

```

Listing 5.2: oneof arbitrary constraint

```

1 arbitrary(listof(L), X, Vars) :-
2   ground(L), length(L, N),
3   arbitrary(listof(L, 1, N), X, Vars).
4
5 arbitrary(listof(L, Size), X, Vars) :-
6   arbitrary(listof(L, Size, Size), X, Vars).
7
8 arbitrary(listof(L, Min, Max), X, Vars) :-
9   ground(L), length(L, N),
10  Min >= 0, Max <= N, Size in Min..Max,
11  length(Idxs, N), domain(Idxs, 1, N), all_distinct(Idxs),
12  Vars = [Size|Idxs],
13  when(ground(Vars), arbitrary_instantiate_listof(L, Idxs, Size, X)).
14
15 arbitrary_instantiate_listof(_, _, 0, []).
16 arbitrary_instantiate_listof(L, [Idx|Idxs], Size, [X|Xs]) :-
17  S1 is Size - 1, nth1(Idx, L, X), arbitrary_instantiate_listof(L, Idxs, S1, Xs).

```

Listing 5.3: listof arbitrary constraint

The `oneof` and `listof` arbitraries can then be used to create more complex arbitraries by creating a wrapper that calls these arbitraries with specific parameters. For example, the name of a person or a country can be described by defining a database with possible values and using `oneof` to pick a random one (Listing 5.4).

```

1 first_name('Michael').
2 ...
3 first_name('Tanya').
4
5 last_name('Smith').
6 ...
7 last_name('Morrison').
8
9 %country(CountryName, Continent)
10 country('Afghanistan', asia).
11 ...
12 country('Zimbabwe', africa).
13
14 arbitrary(name, X, Vars) :-
15     arbitrary(oneof(first_name), FN, FNVars),
16     arbitrary(oneof(last_name), LN, LNVars),
17     append(FNVars, LNVars, Vars),
18     when(ground(Vars), (
19         atom_concat(FN, ' ', Aux),
20         atom_concat(Aux, LN, X)
21     )).
22
23 arbitrary(first_name, X, Vars) :-
24     arbitrary(oneof(first_name), X, Vars).
25
26 arbitrary(last_name, X, Vars) :-
27     arbitrary(oneof(last_name), X, Vars).
28
29 arbitrary(country, X, Vars) :-
30     arbitrary(oneof(country/2), X, Vars).
31 arbitrary(country(Continent), X, Vars) :-
32     arbitrary(oneof(country/2, [Continent]), X, Vars).

```

Listing 5.4: Name and Country arbitraries

5.2.2 Composite 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. This predicate is used in the *constrain* phase to recursively call the *constrain* function on each of the components that make up the whole composite type, therefore constraining it. It is worth noting that the difference between `arbitrary/3` and `composite/3` is that `composite/3` is constrained based on other already defined types (`arbitrary` or `composite`) but `arbitrary/3` is constrained based on the specific constraints defined in the predicate which can call other `arbitrary` predicates but doesn't need to. Listing 5.5 contains examples of composite types as defined by their base types.

```

1 composite(book,
2     [Title:book_title, AuthorID:id, YearOfRelease:integer(100, 2026), Pages:integer
3       (20, 1000), Genres:book_genres],
4     book(Title, AuthorID, YearOfRelease, Pages, Genres)).
5
6 composite(author,
7     [AuthorID:id, Name:name, YearOfBirth:integer(100, 2026), CountryOfBirth:country
8       ],
9     author(AuthorID, Name, YearOfBirth, CountryOfBirth)).

```

Listing 5.5: Example of composite constraints

5.2.3 Constrainer

The listings from the previous subsections are executed by the `constrain_arguments/3` predicate that, as the name suggests, constrains the arguments based on their associated type. The predicates in Listing 5.6 are responsible for supporting all the previously mentioned arbitrary and composite types, as well as for the recursive nature of the constraints.

```

1 % constrains argument for simple arbitrary types
2 constrain_by_type(Type, Var, [Var:Type], [Var]) :-
3     arbitrary(Type, Var).
4
5 % constrains argument for arbitraries that are defined by auxiliary variables
6 constrain_by_type(Type, Value, [Value:Type], Vars) :-
7     arbitrary(Type, Value, Vars).
8
9 % constrains argument for composites of one arbitrary type
10 constrain_by_type(Type, Value, [Value:Type|ValuesWithBaseTypes], Vars) :-
11     % verify there is a composite of the given type
12     clause(composite(Type, Var:BaseType, Value), ArgConstraints),
13     constrain_by_type(BaseType, Var, ValuesWithBaseTypes, Vars),
14     ArgConstraints.
15
16 % constrains arguments for composites of many arbitrary types
17 constrain_by_type(Type, Value, [Value:Type|ValuesWithBaseTypes], Vars) :-
18     % verify there is a composite of the given type
19     clause(composite(Type, [HBaseTypes|TBaseTypes], Value), ArgConstraints),
20     constrain_arguments([HBaseTypes|TBaseTypes], ValuesWithBaseTypes, Vars),
21     ArgConstraints.
22
23 constrain_list_by_element_type(Type, List, ValuesWithBaseTypes, Vars) :-
24     maplist(constrain_by_type(Type), List, ValuesWithBaseTypes, LVars),
25     append(LVars, Vars).
26

```

```

27 constrain_argument(Var:Type, ValuesWithBaseTypes, Vars) :-
28     constrain_by_type(Type, Var, ValuesWithBaseTypes, Vars).
29
30 constrain_arguments(Args, ValuesWithBaseTypes, Vars) :-
31     maplist(constrain_argument, Args, ValuesWithBaseTypes, LVars),
32     append(LVars, Vars).

```

Listing 5.6: Constrain arguments function

It's worth reinforcing that these are purely constrainers. The generation is then done by the generator, explained later in section 5.4, which uses the variables returned by these constrainers.

5.3 Database Generation with CLP

In addition to the aforementioned `arbitrary` and `composite` constrainers, there is the `database` constraider that, as the name suggests, is used to constrain databases that, after being generated, are going to be asserted into the knowledge base. The development of this constraider was the part that took the most development time, and as such, many iterations of this constraider were designed and developed, eventually leading to the final implementation explained in the next subsection (Section 5.3.1). For more details of the development process and why some decisions were made, check Section 5.3.2.

5.3.1 Final Implementation

The final implementation of the database consists of a stage-based *constrain-and-generate* approach. The reasons for this choice are explained in the development process subsection (Section 5.3.2).

A database definition is a series of `stage` statements defined by a constraint body and a list of variables that must be defined before the next stage is executed, and it ends with a `items` statement that defines where the final items that will be asserted to the knowledge base are. Listing 5.7 shows an example of such a database definition.

```

1 database(books_and_authors, books_and_authors(book(10,25), author(5,10))).
2 database(books_and_authors(book(B1, B2), author(A1, A2))) :-
3     stage((NBooks in B1..B2, NAuthors in A1..A2, NBooks #> NAuthors),
4           [NBooks, NAuthors]),
5     stage((
6         database_primary_key(AuthorIDs, NAuthors),
7         database_foreign_key(one_to_many, AuthorIDs, BookAuthorIDs, NBooks),
8         nvalue(NAuthors, BookAuthorIDs)
9     ), [AuthorIDs, BookAuthorIDs]),
10    stage((
11        length(Authors, NAuthors),

```

```

12     constrain_list_by_element_type(author, Authors, _AuthorsWithBaseTypes,
13         AuthorVars),
14     get_args(Authors, [AuthorIDs, _AuthorNames, AuthorYears, _AuthorCountries])
15     ,
16     length(Books, NBooks),
17     unconstrained_list_by_element_type(book, Books, BookBaseTypes),
18     get_args(Books, [BookTitles, BookAuthorIDs, BookYears, BookPages,
19         BookGenres]),
20     BookBaseTypes = [_TitleType, AuthorIDType, YearsType, PagesType, GenresType
21         ],
22     constrain_list_by_element_type(AuthorIDType, BookAuthorIDs, _,
23         BookAuthorIDVars),
24     constrain_list_by_element_type(YearsType, BookYears, _, BookYearsVars),
25     constrain_list_by_element_type(PagesType, BookPages, _, BookPagesVars),
26     constrain_list_by_element_type(GenresType, BookGenres, _, BookGenresVars),
27     constrain_by_type(book_titles(NBooks), BookTitles, _, BookTitlesVars),
28
29     user:book_years_const(AuthorIDs, AuthorYears, BookAuthorIDs, BookYears)
30     ), [AuthorVars, BookAuthorIDVars, BookYearsVars, BookPagesVars, BookGenresVars,
31         BookTitlesVars]),
32     items([Authors, Books]).
33
34 book_years_const([], [], _, _).
35 book_years_const([AuthorID|AIDs], [AuthorYear|AYears], BookAuthorIDs, BookYears) :-
36     book_years_const_aux(AuthorID, AuthorYear, BookAuthorIDs, BookYears),
37     book_years_const(AIDs, AYears, BookAuthorIDs, BookYears).
38
39 book_years_const_aux(_, _, [], []).
40 book_years_const_aux(AuthorID, AuthorYear, [BookAuthorID|BAuthorIDs], [BookYear|
41     BYears]) :-
42     (BookAuthorID #= AuthorID) #=> (AuthorYear + 15 #=< BookYear #/\ AuthorYear +
43         125 #>= BookYear),
44     book_years_const_aux(AuthorID, AuthorYear, BAuthorIDs, BYears).

```

Listing 5.7: Books and Authors Database Example

In Listing 5.7, the predicate `database/1` is not a predicate that is executed, but instead can be thought of as a declarative *"recipe"* that describes the stages that constrain and then the variables that need to be defined before the next stage, and the items it produces. It consists of stage statements and a final `items` statement. Stage statements have 2 arguments. The first is a constraint body that consists of normal Prolog and CLP instructions that constrain the database. The second is a list of all the variables that must be defined before continuing to the next stage, which can have recursive lists of variables. The final `items` statement defines the variables that contain the final items that are going to be asserted into the knowledge base.

Database aliases can be created to ease the creation of databases with the help of the `database/2` predicate. This predicate has two arguments: the first is the alias, and the second is

the database it represents. These aliases are not recursive. Because of this, the value in the second position must be a valid database, i.e., have a `database/1` predicate defined for that value. We can see an example of this in the first line of the previous listing (Listing 5.7).

The interpreter of this database *"recipe"* is the `constrain_database/3` predicate with the help of the `constrain_database_parser/3` predicate. The code for these predicates is shown in Listing 5.8.

```

1 constrain_database(Database, Items, Vars) :-
2     (database(Database, Alias) -> DatabaseSpec = Alias ; DatabaseSpec = Database),
3     clause(database(DatabaseSpec), DatabaseBody),
4     constrain_database_parser(DatabaseBody, Items, Vars).
5
6 constrain_database_parser(DatabaseBody, Items, Vars) :- constrain_database_parser(
7     DatabaseBody, Items, Vars, [], 1).
8
9 constrain_database_parser(items(ListofListsofItems), Items, [], _PrevVars,
10    LastStageCheck) :- !,
11    when(ground(LastStageCheck), append(ListofListsofItems, Items)).
12
13 constrain_database_parser((stage(StageBody, StageVars), T), Items, [StageVars |
14    NextVars], PrevVars, LastStageCheck) :- !,
15    is_list(PrevVars),
16    when(ground([LastStageCheck | PrevVars]), (StageBody, StageCheck = 1)),
17    constrain_database_parser(T, Items, NextVars, StageVars, StageCheck).
18
19 constrain_database_parser(_, _, _, _, _) :- !, write('A database definition should
20    be in the form of {(stage(Body, Vars),)* items(L, Items)}').

```

Listing 5.8: Database Predicate Parser

In Listing 5.8, the predicate `constrain_database` resolves the aliases, if there are any, and checks if there is a `database/1` definition for the given value and calls the parser. The `constrain_database_parser` is the one responsible for actually parsing the stages and items, which sets up listeners for when the previous stage ends to run the next stage with the `when/2` predicate, and returns `Items`, which is a variable that will contain every item to be asserted into the knowledge base. One important thing this predicate also does while parsing stages is the *stage check* that adds a variable that verifies that the previous stage has been executed, this prevents for example, stage 3 executing before stage 2 if, for some reason, stage 2 variables were ground before the variables from stage 1, which would break the trust that every previous stages' variables are ground.

It's important to note that the use of this *stage-based* strategy makes it so that the CSP is never fully defined at any given moment. This means that the use of algorithms, such as the one described in the paper Ziat and Pépin [2025], for the generation of the databases becomes very difficult. Also, this strategy assumes that in the generation process, every possible value from the previous states has a solution in the current one, so one must be careful when designing a database constrainer.

In Listing 5.9, a simple database is defined that, given two numbers A and B, instantiates one item where the first element is greater than or equal to the second element. In this implementation, when generating the values, the generator will go in stage order, so X will be defined first, and only after X is defined will Y be defined. After the first stage, the value of X can't change for the next stages, so either there's a solution or the generation fails. This means that, for example, if in line 3 the `X #>= Y` expression was `X #> Y`, there's a possibility that X is instantiated as the lowest value A and then Y has no solutions. This responsibility lies with the developer of the database constrainer. To fix this, either join both stages or guarantee that there are solutions by, for example, changing stage 1 to be `X in (A+1) .. B`.

```

1 database(simple(A, B)) :-
2     stage((X in A..B), [X]),
3     stage((Y in A..B, X #>= Y), [Y]),
4     items([ints(X, Y)]).
```

Listing 5.9: Simple Database Example

5.3.2 Development Process

The initial implementation was based on a *constrain-and-generate* approach, where we have a `database/3` predicate that defines the constraints of the database and returns the domain variables that need to be set to finalize the items, as well as the variable that will contain the items themselves. Then, when it comes time to generate the database, all we need to do is give random feasible values to the domain variables and to assert the items into the knowledge base.

The initial idea was to eventually implement the algorithm described in the paper by Ziat and Pépin [2025] because of its efficiency and uniformity for the generation process, but soon after starting to develop an actual solution, it became very clear that, if the goal was to support all CLP constraints that are native to SICStus Prolog's `library(clpfd)` to constrain the items of the database themselves, some variables needed to be set so that other constraints could be set. This happens because the problem of generating a database is analogous to the problem of generating a list of elements and then *asserting* the elements into the knowledge base. This means that we need to be able to have lists of variable size and be able to apply constraints to them at the same time. For example, to generate a database consisting of 2 composite types, `book` and `author` (Listing 5.10), the first step is to generate the lists of books and authors with the correct size and then apply constraints to their elements.

```

1 composite(book,
2     [Title:book_title, AuthorID:id, YearOfRelease:integer(100, 2026), Pages:integer
3     (20, 1000), Genres:book_genres],
4     book(Title, AuthorID, YearOfRelease, Pages, Genres)).
```

```

5 composite(author,
6   [AuthorID:id, Name:name, YearOfBirth:integer(100, 2026), CountryOfBirth:country
7     ],
  author(AuthorID, Name, YearOfBirth, CountryOfBirth)).

```

Listing 5.10: Composite Types for Books and Authors Database

```

1 database(books_and_authors(book(B1, B2), author(A1, A2)), Items, Vars) :-
2   NBooks in B1..B2, NAuthors in A1..A2, NBooks #> NAuthors,
3
4   length(Books, NBooks), length(Authors, NAuthors),
5   constrain_list_by_element_type(author, Authors, _AuthorsWithBaseTypes,
6     AuthorVars),
7   constrain_list_by_element_type(book, Books, _BooksWithBaseTypes, BookVars),
8   get_args(Authors, [AuthorIDs, _AuthorNames, AuthorYears, _AuthorCountries]),
9   get_args(Books, [_BookTitles, BookAuthorIDs, BookYears, _BookPages, _BookGenres
10     ]),
11
12   domain(AuthorIDs, 1, NAuthors), all_distinct(AuthorIDs),
13   domain(BookAuthorIDs, 1, NAuthors), nvalue(NAuthors, BookAuthorIDs),
14   book_years_const(AuthorIDs, AuthorYears, BookAuthorIDs, BookYears),
15
16   append(AuthorVars, BookVars, LVars),
17   Vars = [NBooks, NAuthors | LVars],
18   append(Books, Authors, Items).

```

Listing 5.11: Books and Authors Database Naive Example

In Listing 5.11, we start by defining the domain variables `NBooks` and `NAuthors` that define the number of books and authors, respectively, and then call the `length/2` predicate to get actual lists `Books` and `Authors`. This is already a problem, because the `length/2` predicate is not part of `library(clpfd)`, which means that calling `length(Books, NBooks)`, for example, causes `NBooks` to be instantiated to the lowest possible value. This is not what we want. We need to keep all the variables that define the problem as variables until the generation process. My first solution to this problem was the creation of the `constrained_length/3` predicate.

```

1 % constrained_length(+Length, ?List, -InConstraints).
2 constrained_length(Length, List, InConstraints) :-
3   fd_max(Length, MaxLength, !, MaxLength \= sup,
4     length(List, MaxLength), length(InConstraints, MaxLength),
5     constrained_length(List, Length, InConstraints, 0).
6 constrained_length([], _, [], _).
7 constrained_length([_ | Xs], Length, [C|Cs], Idx) :-

```

```

8      C #<=> Idx #< Length, I1 is Idx + 1, constrained_length(Xs, Length, Cs, I1).

```

Listing 5.12: Constrained Length Predicate

The `constrained_length/3` predicate, described in Listing 5.12, constrains a variable `List` with the given `Length`, which is a bounded domain variable, and returns the `InConstraints`, which is a list of the same size as `List` but with only 1s and 0s that represent whether or not that element is in the final list or not, respectively. This, however, doesn't fully replace the `length/2` predicate, because now, to apply a constraint such as `all_distinct`, we must only consider the elements that are actually in the list using `InConstraints` or alternatively using the `Length` domain variable. This solution, however, requires writing a new predicate that applies the constraints only to the correct part of the list.

```

1  % all_distinct_constrained_length_list(+List, +InConstraints).
2  all_distinct_constrained_length_list([], []).
3  all_distinct_constrained_length_list([X | Xs], [_ | Cs]) :-
4      all_distinct_constrained_length_list(Xs, [X], Cs).
5
6  all_distinct_constrained_length_list([], _, []).
7  all_distinct_constrained_length_list([X | Xs], Ys, [C | Cs]) :-
8      all_distinct_constrained_length_list_aux(X, Ys, C),
9      all_distinct_constrained_length_list(Xs, [X | Ys], Cs).
10
11 all_distinct_constrained_length_list_aux(_, [], _).
12 all_distinct_constrained_length_list_aux(X, [Y | Ys], C) :-
13     X #\= Y #<= C, all_distinct_constrained_length_list_aux(X, Ys, C).

```

Listing 5.13: All Distinct for Constrained Length Lists Predicate

In Listing 5.13, the `all_distinct_constrained_length_list/2` predicate is defined; it creates `X #\= Y` constraints only for the elements that have their `InConstraint` value at 1. This approach brings 2 problems: first, this is inefficient by not using the proper native `all_distinct/1` predicate, and instead creates hundreds of single inequality constraints; second, this is not scalable at all and requires a new specialized predicate, such as this one, for every single constraint available in `library(clpfd)`. With this in mind, this approach was not deemed viable, and so a new approach was devised.

The solution to the "length" problem is to define some variables before defining constraints dependent on those variables in a way that allows for the use of native CLP predicates. This requires creating a *stage-based* strategy that ensures some variables are defined before constraints that depend on them are set. And so, after trying many different ways, what resulted was the final implementation (Listings 5.7 & 5.8) explained in detail in the previous subsection (Section 5.3.1).

5.4 Generator

There are 2 predicates for the generation of the values for arguments and for databases, `generate_arguments` and `generate_databases` respectively, both of which divide the variables into independent **CSPs** with the `divide_csps/2` (Listing 5.14) and then call the method `generate_single_csp/2` (Listings 5.15 and 5.16) for every single one of those **CSPs**.

```

1 divide_csps(Vars, CSPs) :-
2     divide_csps_get_var_ord_set(Vars, VarsOrdSet),
3     divide_csps_aux(VarsOrdSet, CSPs).
4 divide_csps_aux([], []).
5 divide_csps_aux(VarsOrdSet, [CSPVars|CSPs]) :-
6     [Var|_] = VarsOrdSet,
7     fd_closure([Var], CompleteCSP),
8     ord_intersection([VarsOrdSet, CompleteCSP], CSPVars),
9     ord_subtract(VarsOrdSet, CSPVars, RestVarsOrdSet),
10    divide_csps_aux(RestVarsOrdSet, CSPs).
11
12 divide_csps_get_var_ord_set(Vars, VarsOrdSet) :-
13     divide_csps_get_var_ord_set(Vars, [], VarsOrdSet).
14 divide_csps_get_var_ord_set([], Acc, VarsOrdSet) :- list_to_ord_set(Acc, VarsOrdSet
15 ).
16 divide_csps_get_var_ord_set([Var|Vars], Acc, VarsOrdSet) :-
17     ground(Var), !, divide_csps_get_var_ord_set(Vars, Acc, VarsOrdSet).
18 divide_csps_get_var_ord_set([Var|Vars], Acc, VarsOrdSet) :- is_list(Var), !,
19     append(Var, Vars, NewVars), divide_csps_get_var_ord_set(NewVars, Acc,
20     VarsOrdSet).
21 divide_csps_get_var_ord_set([Var|Vars], Acc, VarsOrdSet) :-
22     divide_csps_get_var_ord_set(Vars, [Var|Acc], VarsOrdSet).

```

Listing 5.14: CSPs Divider

In listing 5.14, the predicate `divide_csps/2` receives a list of variables and returns a list of lists of variables, with each one of the lists being an independent **CSP**. It does this by using the predicate `fd_closure/2`, which, given a set of variables, returns their closure or, in other words, the independent **CSP** they're part of. This predicate does this efficiently by the use of the library `ordsets`, to get only the relevant variables out of the full closure and to remove the variables from the current **CSP** from the full set of variables. Because the full closure might contain auxiliary variables that we didn't create, it's important to filter the variables from the closure to only those created by the user.

To do the actual generation, there are two rejection sampling methods: one that returns a uniform distribution of every value in the solution space and one that isn't uniform. For both methods, we iterate through every variable and choose a random value from the possible values

and set that value to the variable. This is done with the `random_variable_value/2` predicate (Listing 5.17). The difference comes in the order of operations. The uniform one chooses the value and continues to the next variable before setting the variable, and the non-uniform one sets the variable before continuing to the next one. This subtle change is the difference between good performance and uniformly distributed values. Listing 5.15 shows the non-uniform rejection sampling, and Listing 5.16 shows the uniform rejection sampling.

```

1 generate_single_csp(rejection_sampling, Vars) :-
2     % rejection sampling
3     (   random_checked_labeling(Vars)
4     -> true
5     ;   generate_single_csp(rejection_sampling, Vars)).
6
7 random_checked_labeling([]).
8 random_checked_labeling([Var|Vars]) :-
9     ground(Var), !,
10    random_checked_labeling(Vars).
11 random_checked_labeling([Var|Vars]) :-
12    random_variable_value(Var, Value),
13    % do the recursive call after constraining the variable to improve performance
14    % ( doesn't give a uniform distribution of random values )
15    Var #= Value, random_checked_labeling(Vars).

```

Listing 5.15: Rejection Sampling

```

1 generate_single_csp(uniform_rejection_sampling, Vars) :-
2     % uniform rejection sampling
3     (   random_unchecked_labeling(Vars)
4     -> true
5     ;   generate_single_csp(uniform_rejection_sampling, Vars)).
6
7 random_unchecked_labeling([]).
8 random_unchecked_labeling([Var|Vars]) :-
9     ground(Var), !,
10    random_unchecked_labeling(Vars).
11 random_unchecked_labeling([Var|Vars]) :-
12    random_variable_value(Var, Value),
13    % do the recursive call before constraining the variable because then the
14    % values are chosen,
15    % otherwise, it is not uniform sampling because it biases towards the correct
16    % solutions
17    random_unchecked_labeling(Vars), Var #= Value.

```

Listing 5.16: Uniform Rejection Sampling

```

1 random_variable_value(Var, Value) :- !,
2   fd_set(Var, Set), fdset_size(Set, Size), !, Size \= sup,
3   random(0, Size, Idx), nth0_fdset(Set, Idx, Value).
4 random_variable_value(Var, Value) :- % old version
5   fd_set(Var, Set), fdset_to_list(Set, List),
6   random_member(Value, List).
7
8 nth0_fdset(Set, Idx, Value) :- Idx >= 0,
9   fdset_parts(Set, Min, Max, Rest),
10  NValues = Max - Min + 1,
11  ( Idx < NValues -> Value is Min + Idx
12  ; NextIdx is Idx - NValues, nth0_fdset(Rest, NextIdx, Value)).

```

Listing 5.17: Random Variable Value Predicate

In the case of database generation, the uniformity of the values, even when using the uniform rejection sampling generation, is put into question because of the *stage-based* strategy, because there is almost certainly a different number of solutions for a given stage, depending on the previous stage's values. The algorithm of the paper by Ziat and Pépin [2025] was not implemented because, by having a *stage-based* strategy, the CSP is never fully represented at a given point, and even if it was, the CSP changes depending on the values chosen, which means that the work to create the generator for a given test would be wasted because in the next test it would be a different CSP.

Despite the use of rejection sampling, CLP is still being used to optimize this process. If the default rejection sampling is used (non-uniform), after each value is set, the constraint solver constrains the search space for the given value. This means that if an infeasible value is set, the generation will fail and try again without generating all other variables. If uniform rejection sampling is used, then CLP is only used at the end to check the solution.

5.5 Testing Interface

In the developed library, a property is defined as any predicate that takes zero or more arguments, each with its own type. The specification of the type of each argument with the `:<type>` syntax is required because that's the type that the constrainers and the shrinker will ultimately use to constrain and shrink that variable, respectively. Listing 5.18 illustrates such a property.

```

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

```

Listing 5.18: Property Example

To define a custom arbitrary or composite type, one must give the `:- multifile` directive and then the corresponding implementation. Listing 5.19 provides an example of such a definition for an arbitrary predicate.

```

1 :- multifile arbitrary/3.
2 arbitrary(city_name, X, Vars) :-
3     arbitrary(oneof([lisbon, porto, napoli, lyon, london, tokyo, berlin]), X, Vars)
4     .

```

Listing 5.19: Custom arbitrary Example

To define a custom database, again, one must give the `:- multifile` directive before and its implementation. Listing 5.20 shows a simple example of a custom database definition.

```

1 :- multifile database/1.
2
3 database(test_db(A, B)) :-
4     stage((X in A..B), [X]),
5     items([test_pred(X)]).

```

Listing 5.20: Custom Database Example

To execute a property test, one must use the main predicate `quickcheck` with the name of the property and the arity of the property it's testing. A list of running options can be given with the following possible values:

- `database(DatabaseName)`: defines that a database with that name exists and will be generated before each test.
- `argument_gen(Method)`: defines the generation algorithm used for arguments. The possible values are `rejection_sampling` (the default) and `uniform_rejection_sampling`.
- `database_gen(Method)`: defines the generation algorithm used for databases. The possible values are `rejection_sampling` (the default) and `uniform_rejection_sampling`.

In addition to that, the number of tests executed for a given property test can be modified as well. The default number of tests is 100. Listing ?? shows the many different ways to use the `quickcheck` predicates and their meanings.

```

1 % Basic Property Test
2 ?- quickcheck(prop_dec2bin_power/1).
3 100 tests OK
4

```

```

5 % Custom number of tests
6 ?- quickcheck(prop_dec2bin_power/1, 1200).
7 1200 tests OK
8
9 % Custom database (other options are given in the same list)
10 % (other databases can be defined no just only one)
11 ?- quickcheck(prop_dec2bin_power/1, [database(test_db(1,100))]).
12 100 tests OK
13
14 % Both options
15 ?- quickcheck(prop_dec2bin_power/1, [database(test_db(1,100))], 800).
16 800 tests OK

```

Listing 5.21: Quickcheck Predicate Examples

5.6 Limitations

This library, despite being as generic and reusable across a wide range of Prolog testing scenarios, has three main limitations in its implementation: handling tests that involve input/output, shrinking the database values, and targeting values that exist in the database.

Handling input/output was already a limitation of the tool from the previous work, and because it was not the focus of this dissertation, it remains. This limitation refers to predicates that use I/O operations, such as `read` and `write`, and the incapability of the tool to simulate and/or capture these operations. For example, in Listing 5.22, 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 5.22: Input/Output Predicate Example

The second limitation is much more relevant to the context of this dissertation. The *stage-based* strategy for database constraining and generation causes shrinking in any meaningful way to become very cumbersome to try and implement because it would require saving all the constraints applied to then keep the integrity of the database. This is due to constraints of later stages being completely invalidated once a previous stage's values change. This causes, for example, if we reduce the size of the database, then the constraints dependent on those variables need to be set again, but this most of the time means a regeneration of the database as a whole, which, as a shrinking agent, isn't very effective because it's likely that the error-causing property of the

database will be lost by a regeneration of the database. Shrinking by the removal of an element from the database would also be very cumbersome because, as previously stated, there is no good way to check the integrity of the constraints of the database after removing one or more elements. This means that the whole database would have to be shown to the user with no shrinking, which is not ideal for finding bugs.

The third limitation refers to the creation of properties that receive arguments that are in some way part of the database because there is no good way to bias those arguments to be part of the database. For example, in Listing 5.23, the predicate `prop_similarity/2` receives two book titles and tests the predicate `similarity/3` by comparing it with a correct solution `pfl_similarity/3`, and they are random values from all the possible book titles. This means that if the database has few elements, for example, there is a big chance that at least one of the titles of the books are not in the database, which would cause most tests to fail because both predicates fail, which is correct but doesn't test the good path very often. A workaround is to create custom databases and arbitraries that give more guarantees, but this is more work for the developer of the tests, which is not ideal. Ideally, one would have a way to describe an argument that is guaranteed to be in the database, but this would require databases to be generated first, which they are not, and then something like an `arbitrary(oneof...)` would be used to get one from the possible values.

```

1 prop_similarity(Title1:book_title, Title2:book_title) :-
2     (pfl_similarity(Title1, Title2, SimPFL), similarity(Title1, Title2, Sim),
3       SimPFL == Sim);
4     (\+ pfl_similarity(Title1, Title2, _SimPFL), \+ similarity(Title1, Title2, _Sim
5       )).

```

Listing 5.23: Database Arguments Example

5.7 Summary

This chapter describes how the new and updated components integrate **CLP** to support the generation of complex inputs, particularly databases and structured data. The implementation follows a *constrain-and-generate* approach, where previously existing generators were transformed into constrainers, database constrainers were developed, and a standalone generator is the only component capable of generating the final values. To address the challenges of generating databases with interdependent components, a stage-based architecture was developed. This approach allows constraints to be incrementally defined while still leveraging native **CLP** predicates available in SICStus Prolog. As for the generation process, strategies based on rejection sampling were implemented, together with a way to separate independent **CSPs** to improve efficiency.

Regarding RQ1, the results show that input generation can be achieved by modelling complex structures as declarative constraints and separating the constraining and generation phases. Composite data types and databases can be represented as a **CSP**, with a stage-based strategy enabling the creation of dynamic structures whose size and constraints depend on the previously generated values.

Regarding RQ2, this chapter demonstrates that **CLP** is a suitable and effective approach for input generation. **CLP** provides a declarative way to express relationships, integrity constraints, and dependencies between generated values while allowing the reuse of native SICStus Prolog constraint-solving capabilities.

By the end of the chapter, a database generation extension to the previously developed Prolog framework was defined. The implementation allows for the definition of databases with intricate constraints that are generated for each **PBT** test, but further testing is necessary to evaluate its usefulness and efficiency. The next chapter focuses on the application of the developed approach to real-world cases and on identifying its advantages and disadvantages.

Chapter 6

Experiment and Results

This chapter presents the experiments executed to assess the usability and performance of the developed approach. The first section contains examples of databases and their implementation using that approach, and compares it with a normal unit test scenario, as well as highlighting its performance metrics. The second section gathers all the results from the previous section and presents the advantages and disadvantages of this approach.

6.1 Setup

To test the developed approach, a qualitative and quantitative comparison between **UT** and **PBT** was conducted. For this comparison, the professors provided the questions, solutions, as well as anonymized student responses to the exam for the PFL course in 2024. Using real-world data allows for a more correct assessment of the impact that the developed approach would have if it were used for grading real tests.

Because of the amount of manual effort that developing database constrainers, properties, and even shrinkers for every exercise in the exam entails, one exercise was chosen to showcase the framework in action. This exercise was chosen because: it represents a typical programming problem that uses databases and relies on the relationships between different items of the database; it has 200+ valid responses for each testing type; and it doesn't require anything that is not currently supported by the framework.

The Python testing framework developed by the previous work [de Sá, 2025] was modified to run on a Windows OS, as well as to execute the code in the right order so that database definitions work. Figure 6.1 presents a high-level overview of the architecture used and the data flow that was required, which remains the same as the previous work.

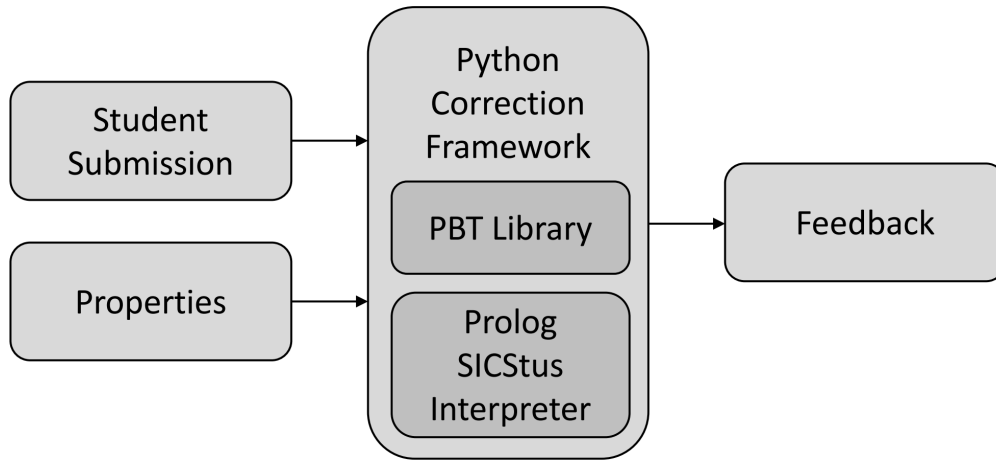


Figure 6.1: Overview of the Python testing library workflow (extracted from [de Sá, 2025])

For the selected question, both the unit tests and the property-based tests were executed for each student. The result of each test, as well as the execution time were recorded by the Python framework. All property-based tests were performed with 100 tests, which is the default. Shrinking was enabled but inconsequential because the types used had no shrinking processes defined.

Every test was run on a Windows 11 25H2 machine with an AMD Ryzen 7 3700X 8-Core Processor (16 logical cores) at 3.60GHz and 32GB of RAM, L1 cache of 512KB, L2 cache of 4MB, and L3 cache of 32MB. The SICStus Prolog interpreter version was 4.10.1, which was the latest available at the time the requirements were gathered, and Python 3.12.2 was used.

Despite execution time not being a good metric to compare **UT** and **PBT** due to the fundamentally different approaches of both test types, it was recorded anyway to provide a general sense of the time that it would take to use each method in a real-world scenario, e.g., to evaluate the whole exam.

6.2 Database Example

This section contains examples of database implementations using the developed approach. Each subsection starts by describing the database and its constraints, followed by the implementation and the unit test counterpart, and ends with the performance metrics of that specific database.

These databases rely on arbitrary constraints which might have to be created by the developer of the database conainer. Their implementations are not going to be shown because most are trivial.

6.2.1 Books and Authors Database

The "Books and Authors" database is composed of the composite types `book` and `author` presented in Listing 6.1, which in depend on the arbitraries `book_title`, `id`, `integer(A, B)`,

book_genres, name, and country. The arbitraries book_title, name, and country pick one value from a list of possible values or from a predicate in the knowledge base. The arbitrary book_genres forms a list with one or more elements from another list. The arbitrary integer(A, B) picks a value between A and B inclusive. The arbitrary id is equivalent to a positive integer.

```

1 composite(book,
2   [Title:book_title, AuthorID:id, YearOfRelease:integer(100, 2026), Pages:integer
3     (20, 1000), Genres:book_genres],
4   book(Title, AuthorID, YearOfRelease, Pages, Genres)).
5 composite(author,
6   [AuthorID:id, Name:name, YearOfBirth:integer(100, 2026), CountryOfBirth:country
7     ],
8   author(AuthorID, Name, YearOfBirth, CountryOfBirth)).

```

Listing 6.1: Books and Authors Composite Types

For the **PBT** tests, the predicate `database(books_and_authors(book(B1, B2), author(A1, A2)))` (Listing 6.2) was created. This predicate is responsible for constraining the "Books and Authors" database. It sets the constraints between author IDs and book IDs (primary key to foreign key relationship), constrains book titles to be unique, and guarantees that any given book is published between the author's 15th and 125th birthday.

```

1 :- dynamic book/5, author/4.
2
3 database(books_and_authors, books_and_authors(book(10,25), author(5,10))).
4 database(books_and_authors(book(B1, B2), author(A1, A2))) :-
5   stage((NBooks in B1..B2, NAuthors in A1..A2, NBooks #> NAuthors),
6     [NBooks, NAuthors]),
7   stage((
8     database_primary_key(AuthorIDs, NAuthors),
9     database_foreign_key(one_to_many, AuthorIDs, BookAuthorIDs, NBooks),
10    nvalue(NAuthors, BookAuthorIDs)
11  ), [AuthorIDs, BookAuthorIDs]),
12   stage((
13     length(Authors, NAuthors),
14     constrain_list_by_element_type(author, Authors, _AuthorsWithBaseTypes,
15       AuthorVars),
16     get_args(Authors, [AuthorIDs, _AuthorNames, AuthorYears, _AuthorCountries])
17   ,
18     length(Books, NBooks),
19     unconstrained_list_by_element_type(book, Books, BookBaseTypes),
20     get_args(Books, [BookTitles, BookAuthorIDs, BookYears, BookPages,
21       BookGenres]),

```

```

19     BookBaseTypes = [_TitleType, AuthorIDType, YearsType, PagesType, GenresType
20         ],
21     constrain_list_by_element_type(AuthorIDType, BookAuthorIDs, _,
22         BookAuthorIDVars),
23     constrain_list_by_element_type(YearsType, BookYears, _, BookYearsVars),
24     constrain_list_by_element_type(PagesType, BookPages, _, BookPagesVars),
25     constrain_list_by_element_type(GenresType, BookGenres, _, BookGenresVars),
26     constrain_by_type(book_titles(NBooks), BookTitles, _, BookTitlesVars),
27
28     user:book_years_const(AuthorIDs, AuthorYears, BookAuthorIDs, BookYears)
29 ), [AuthorVars, BookAuthorIDVars, BookYearsVars, BookPagesVars, BookGenresVars,
30     BookTitlesVars]],
31 items([Authors, Books])).
32
33 book_years_const([], [], _, _).
34 book_years_const([AuthorID|AIDs], [AuthorYear|AYears], BookAuthorIDs, BookYears) :-
35     book_years_const_aux(AuthorID, AuthorYear, BookAuthorIDs, BookYears),
36     book_years_const(AIDs, AYears, BookAuthorIDs, BookYears).
37
38 book_years_const_aux(_, _, [], []).
39 book_years_const_aux(AuthorID, AuthorYear, [BookAuthorID|BAuthorIDs], [BookYear|
40     BYears]) :-
41     (BookAuthorID #= AuthorID) #=> (AuthorYear + 15 #=< BookYear #/\ AuthorYear +
42         125 #>= BookYear),
43     book_years_const_aux(AuthorID, AuthorYear, BAuthorIDs, BYears).

```

Listing 6.2: Books and Authors Database Example

For the **UT** tests, the hard-coded database presented in Listing 6.3 was used.

```

1  %author(AuthorID, Name, YearOfBirth, CountryOfBirth).
2  author(1, 'John Grisham', 1955, 'USA').
3  author(2, 'Wilbur Smith', 1933, 'Zambia').
4  author(3, 'Stephen King', 1947, 'USA').
5  author(4, 'Michael Crichton', 1942, 'USA').
6  author(5, 'James Patterson', 1947, 'USA').
7  author(6, 'Dan Brown', 1964, 'USA').
8
9
10 %book(Title, AuthorID, YearOfRelease, Pages, Genres).
11 book('The Firm', 1, 1991, 432, ['Legal thriller']).
12 book('The Client', 1, 1993, 422, ['Legal thriller']).
13 book('The Whistler', 1, 2016, 384, ['Crime thriller']).
14 book('Rogue Lawyer', 1, 2015, 352, ['Legal thriller']).
15 book('The Runaway Jury', 1, 1996, 414, ['Legal thriller']).
16 book('The Exchange', 1, 2023, 338, ['Legal thriller']).
17
18 book('Carrie', 3, 1974, 199, ['Horror']).

```

```

19 book('Pet Sematary', 3, 1983, 374, ['Horror']).
20 book('The Shining', 3, 1977, 447, ['Gothic novel', 'Horror', 'Psychological horror
    ']).
21 book('Under the Dome', 3, 2009, 1074, ['Science fiction', 'Political']).
22 book('Doctor Sleep', 3, 2013, 531, ['Horror', 'Gothic', 'Dark fantasy']).
23
24 book('Jurassic Park', 4, 1990, 399, ['Science fiction']).
25 book('Prey', 4, 2002, 502, ['Science fiction', 'Techno-thriller', 'Horror', '
    Nanopunk']).
26 book('Next', 4, 2006, 528, ['Science fiction', 'Techno-thriller', 'Satire']).
27 book('Micro', 4, 2011, 424, ['Science fiction', 'Adventure', 'Techno-thriller']).
28
29 book('River God', 2, 1993, 662, ['Historical Fiction', 'Fiction', 'Historical', '
    Egypt', 'Fantasy', 'Adventure', 'Thriller']).
30 book('The Seventh Scroll', 2, 1995, 752, ['Historical Fiction', 'Fiction', 'Egypt',
    'Mystery', 'Adventure', 'Thriller']).
31 book('The Sunbird', 2, 1972, 662, ['Fiction', 'Historical Fiction', 'Adventure', '
    Thriller', 'Africa', 'Historical', 'Action']).
32 book('Those in Peril', 2, 2011, 382, ['Fiction', 'Thriller', 'Adventure', 'Action',
    'Historical Fiction', 'Africa', 'Mystery']).

```

Listing 6.3: Books and Authors Database for Unit Tests Example

6.2.2 Similarity Experiment

The `similarity/3` predicate, given two book titles, returns the Jaccard coefficient, also known as intersection over union (IoU), which is determined by the division between the number of elements of the intersection and the number of elements of the union of the genres of the two books associated with the given titles. An example solution is presented in Listing 6.5.

```

1 similarity(Title1, Title2, Similarity):-
2     shared_genres(Title1, Title2, Intersection),
3     book(Title1, _, _, _, Genres1),
4     book(Title2, _, _, _, Genres2),
5     union(Genres1, Genres2, Union),
6     length(Intersection, LI),
7     length(Union, LU),
8     Similarity is LI / LU.

```

Listing 6.4: Similarity Example Solution

For the **PBT** tests, a singular property named `prop_similarity/2` (Listing 6.5) was defined. This property is the generic case, which works by merely comparing the result of the predicate with the result of a correct predicate. It is also important to use different variables for the similarity to guarantee that the student's answer is run with a variable and not a value.

```

1 :- dynamic prop_similarity/2.
2 prop_similarity(Title1:book_title, Title2:book_title) :-
3     (pfl_similarity(Title1, Title2, SimPFL), similarity(Title1, Title2, Sim),
4       SimPFL == Sim);
5
6 pfl_similarity(Title1, Title2, Similarity):-
7     shared_genres(Title1, Title2, Intersection),
8     book(Title1, _, _, Genres1),
9     book(Title2, _, _, Genres2),
10    pfl_union(Genres1, Genres2, Union),
11    length(Intersection, LI),
12    length(Union, LU),
13    Similarity is LI / LU.
14
15 shared_genres(Title1, Title2, CommonGenres):-
16     book(Title1, _, _, Genres1),
17     book(Title2, _, _, Genres2),
18     common_elements(Genres1, Genres2, CommonGenres).
19
20 common_elements([], _, []).
21 common_elements([H|T], L, [H|R]):-
22     member(H, L), !,
23     common_elements(T, L, R).
24 common_elements(_|T, L, R):-
25     common_elements(T, L, R).
26
27 pfl_union(L1, L2, L3):-
28     append(L1, L2, L),
29     sort(L, L3).

```

Listing 6.5: Books and Authors Generic Property

For the **UT** tests, as mentioned before, the unit test suite used in the PFL exam from 2024 was used. The test suite is presented in Listing 6.6.

```

1 similarity('The Firm', 'The Client', Sim). % 1.0, from the examples
2 similarity('Carrie', 'Pet Sematary', Sim). % 1.0, not in the examples
3 similarity('Prey', 'Next', Sim). % from the examples, 0.4
4 similarity('River God', 'The Seventh Scroll', Sim). % not from the examples, 0.625
5 similarity('River God', Second, Sim). % Only first instantiated
6 similarity('The Associate', Second, Sim). % Only first instantiated, does not exist
7 similarity(First, 'The Seventh Scroll', Sim). % Only second instantiated
8 similarity('The Brethren', 'The Seventh Scroll', Sim). % First book does not exist
9 similarity('The Seventh Scroll', 'The Brethren', Sim). % Second book does not
   exist

```

```

10 similarity('The Associate', 'The Brethren', Sim). % books do not exist
11 findall(F-S-Sim, similarity(F, S, Sim), _L), sort(_L,Sorted). % everything
    uninstantiated
12 similarity('Carrie', 'Pet Sematary', 1.0), !. % all instantiated, single, true
13 similarity('Carrie', 'Pet Sematary', 0.4), !. % all instantiated, single, false
14 similarity('River God', 'The Seventh Scroll', 0.625), !. % all instantiated,
    several, true
15 similarity('River God', 'The Seventh Scroll', 0.4), !. % all instantiated, several
    , false
16 similarity('River God', Second, 0.625). % Only first instantiated, Sim given,
    results
17 similarity('River God', Second, 0.2). % Only first instantiated, Sim given, no
    results
18 similarity(First, 'The Seventh Scroll', 0.625). % Only second instantiated, Sim
    given, results
19 similarity(First, 'The Seventh Scroll', 0.2). % Only second instantiated, Sim given
    , no results
20 similarity(First, Second, 0.625). % Only Sim, results
21 similarity(First, Second, 0.3). % Only Sim, no results

```

Listing 6.6: Books and Authors Unit Test Suite

6.3 Results and Conclusions

This section presents the results of running the tests for both property-based tests and unit tests, and it also discusses the advantages and disadvantages of each method.

Table 6.1 shows the number of valid submissions out of the total number of submissions.

Parameter	Count
Total Submissions	329
Error/Timeout Free Submissions (UT)	204
Error/Timeout Free Submissions (PBT)	236

Table 6.1: Distribution of submission types

Table 6.2 shows the number of tests/properties, valid correction submissions, valid incorrect submissions, and the single test execution average, as well as the total evaluation time.

Parameter	Unit Testing	Property-Based Testing
Number of Tests / Properties	21	1
Number of Valid, Fully Correct Submissions	102	181
Number of Valid Submissions Not Fully Correct	102	55
Single Valid Test Average Execution Time (ms)	175	820
Total Evaluation Time for All Valid Submissions (s)	750	193

Table 6.2: Summary of evaluation statistics

Despite **PBT** only using one property, it was still able to find 55 submissions with errors, but it is clear that the **PBT** implementation is lacking. This is due to the fact that it's hard to make properties for database-dependent exercises. Some of the submissions that **UT** caught that **PBT** didn't are submissions that gave the correct answer but gave it multiple times instead of one. It's hard to describe a property that verifies that.

Despite its shortcomings, the **PBT** implementation was very fast at 820ms per test, which in turn is 100 "sub-tests" with each generating its own unique valid database, still managed to be only $\tilde{4.6}$ x slower than the **UT** counterpart at 175ms per test. Because there were 21 unit tests and only 1 property, **PBT** was $\tilde{4}$ x faster at the total evaluation time. This means that if the **PBT** property suite is improved with similarly performing 3 other properties, the coverage could improve a lot and execute in the same amount of time.

In respect to the development process its debatable which one is better, **PBT** or **UT**. Despite **UT** being simple, every test has to be thought out and written, and every database constraint has to be carefully validated by hand when creating a new database. **PBT**, on the other hand, is arguably easier in the sense that you simply declare the constraints and the constraint solver does the rest, but, as this example shows, it can get quite lengthy. It's also worth mentioning that after building the database constrainer for a single test, a database of arbitrary sizes can be created for each test execution. In conclusion, both methods require effort and care to create good and valid databases for the tests.

6.4 Summary

In this chapter, the **PBT** approach was compared to the **UT** approach, and it's clear that further testing is required to determine which one is really better. Both methods require effort and care to create the databases, but it's unclear which one has the upper hand. **PBT**, despite being more lengthy, has a more diverse set of databases at the end. **UT** has one database, but it's of higher quality.

Chapter 7

Conclusion

This chapter presents the conclusion to the work carried out in this dissertation, the summary of findings, the main contributions of the research, and the future work that can be addressed by subsequent iterations.

7.1 Summary of Findings

This dissertation focused on the problem of efficient input data generation for Property-Based Testing (**PBT**) in Prolog, with a particular focus on the use of Constraint Logic Programming (**CLP**) to support structured data generation. The main objective is to investigate whether **CLP** can improve the robustness and efficiency of input generation for a Prolog-based **PBT** tool while also adding the capacity for **PBT** with data structure inputs and how it compares to **UT**.

Regarding RQ1, the results show that input generation can be achieved by modelling complex structures as declarative constraints and separating the constraining and generation phases. Composite data types and databases can be represented as a **CSP**, with a stage-based strategy enabling the creation of dynamic structures whose size and constraints depend on the previously generated values.

Regarding RQ2, chapter 5 demonstrates that **CLP** is a suitable and effective approach for input generation. **CLP** provides a declarative way to express relationships, integrity constraints, and dependencies between generated values while allowing the reuse of native SICStus Prolog constraint-solving capabilities.

Regarding RQ3, the experiments done in chapter 6 were inconclusive as to how the developed **PBT** approach and **UT** compare, and as such, further testing is required. Despite this, the experiments showed that **PBT** was a viable way to create tests and was able to capture some of the solutions with errors.

Also, due to the research done in chapter 3, this dissertation also condenses the available knowledge about the current techniques of data generation with **CLP** and evaluates their feasibility and usefulness to the problem of generating databases for a **PBT** framework in Prolog.

7.2 Main Contributions

This dissertation presents both theoretical and practical contributions to the field of software testing in the context of constraint logic programming and data generation, specifically through the implementation of a database generator using **CLP** for a **PBT** framework in Prolog.

Theoretically, this dissertation presents the available tools and algorithms that one might use to attack the problem of data generation with **CLP**. Practically, this dissertation presents an extension to the previously developed **PBT** framework [de Sá, 2025], which implements a novel *stage-based* strategy to use **CLP** to do full database generation. The main contribution of this work, regarding the previous work by Fábio [de Sá, 2025], is the implementation of a complex database generation framework capable of handling dependencies between components of the database using arbitrary constraints.

7.3 Future Work

While developing this dissertation, several places of improvement were found. While there are probably hundreds of points of improvement, the main ones are:

- Include I/O stream-based property support
- Add shrinking to databases
- Add predicates to define properties that count solutions
- Add ways to define variants of a main database to allow for better properties later on
- Add predicates that allow generating arguments that are guaranteed to be in the instantiated database
- Better testing to compare to **UT**

Bibliography

- Aguirre, N., Bengolea, V. S., Frias, M. F., and Galeotti, J. P. (2011). Incorporating coverage criteria in bounded exhaustive black box test generation of structural inputs. In Gogolla, M. and Wolff, B., editors, *Tests and Proofs - 5th International Conference, TAP@TOOLS 2011, Zurich, Switzerland, June 30 - July 1, 2011. Proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 15–32. Springer.
- Amaral, C., Florido, M., and Costa, V. (2014). PrologCheck - Property-based testing in Prolog. In *Lect. Notes Comput. Sci.*, volume 8475 LNCS, pages 1–17. Springer Verlag service@springer.de.
- Apt, K. R. (1997). *From logic programming to Prolog*. Prentice Hall International series in computer science. Prentice Hall.
- 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. *Inf. Softw. Technol.*, 74:45–54.
- Carlsson, M. and Mildner, P. (2012). Sicstus prolog - the first 25 years. *Theory Pract. Log. Program.*, 12(1-2):35–66.
- Claessen, K. and Hughes, J. (2000). Quickcheck: a lightweight tool for random testing of haskell programs. In Odersky, M. and Wadler, P., editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM.
- Claessen, K., Hughes, J., Palka, M. H., Smallbone, N., and Svensson, H. (2010). Ranking programs using black box testing. In Zhu, H., Chan, W. K., Budnik, C. J., and Kapfhammer, G. M., editors, *The 5th Workshop on Automation of Software Test, AST 2010, May 3-4, 2010, Cape Town, South Africa*, pages 103–110. ACM.
- Colmerauer, A. and Roussel, P. (1993). The birth of prolog. In Lee, J. A. N. and Sammet, J. E., editors, *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 37–52. ACM.
- de Sá, F. A. (2025). *Towards a Support Tool for Prolog Using Property-Based Testing*. PhD thesis, University of Porto.

- Fioravanti, F., Proietti, M., and Senni, V. (2015). Efficient generation of test data structures using constraint logic programming and program transformation. *J. Log. Comput.*, 25(6):1263–1283.
- Garousi, V., Rainer, A., Jr., P. L., and Arcuri, A. (2020). Software-testing education: A systematic literature mapping. *J. Syst. Softw.*, 165:110570.
- Genesereth, M. R. and Ginsberg, M. L. (1985). Logic programming. *Commun. ACM*, 28(9):933–941.
- Goldstein, H., Cutler, J. W., Dickstein, D., Pierce, B. C., and Head, A. (2024). Property-based testing in practice. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pages 187:1–187:13. ACM.
- Jaffar, J. and Lassez, J. (1987). Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 111–119. ACM Press.
- Jamil, M. A., Arif, M., Abubakar, N. S. A., and Ahmad, A. (2016). Software Testing Techniques: A Literature Review. In *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, pages 177–182, Jakarta, Indonesia. IEEE.
- Klammer, C. and Kern, A. (2015). Writing unit tests: It’s now or never! In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–4. IEEE Computer Society.
- Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. (2018). Generating good generators for inductive relations. *Proc. ACM Program. Lang.*, 2(POPL):45:1–45:30.
- Lo, F., Chen, C., and Chen, Y. (2019). Genetic algorithms as shrinkers in property-based testing. In López-Ibáñez, M., Auger, A., and Stützle, T., editors, *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 291–292. ACM.
- Löscher, A. and Sagonas, K. (2017). Targeted property-based testing. In Bultan, T. and Sen, K., editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 46–56. ACM.
- Marriott, K. and Stuckey, P. J. (1998). *Programming with Constraints: An Introduction*. MIT press.
- Milicevic, A., Misailovic, S., Marinov, D., and Khurshid, S. (2007). Korat: A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 771–774. IEEE Computer Society.

- Quadri, S. and Farooq, S. U. (2010). Software Testing – Goals, Principles, and Limitations. *International Journal of Computer Applications*, 6(9):7–10.
- Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41.
- Senni, V. and Fioravanti, F. (2012). Generation of test data structures using constraint logic programming. In Brucker, A. D. and Julliand, J., editors, *Tests and Proofs - 6th International Conference, TAP@TOOLS 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7305 of *Lecture Notes in Computer Science*, pages 115–131. Springer.
- Sullivan, K. J., Yang, J., Coppit, D., Khurshid, S., and Jackson, D. (2004). Software assurance by bounded exhaustive testing. In Avrunin, G. S. and Rothermel, G., editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 133–142. ACM.
- Tarjan, R. E. (1974). A note on finding the bridges of a graph. *Inf. Process. Lett.*, 2(6):160–161.
- Umar, M. A. (2020). A Study of Software Testing: Categories, Levels, Techniques, and Types.
- Verma, A., Department of Computer Science, Amity University, Gurgaon, India, Khatana, A., Department of Computer Science, Amity University, Gurgaon, India, Chaudhary, S., and Department of Computer Science, Amity University, Gurgaon, India (2017). A Comparative Study of Black Box Testing and White Box Testing. *International Journal of Computer Sciences and Engineering*, 5(12):301–304.
- Wallace, M. (2002). Constraint logic programming. In Kakas, A. C. and Sadri, F., editors, *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I*, volume 2407 of *Lecture Notes in Computer Science*, pages 512–532. Springer.
- 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.
- Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2010). Swi-prolog. *CoRR*, abs/1011.5332.
- Ziat, G. and Pépin, M. (2025). An efficient and uniform CSP solution generator generator. In de la Banda, M. G., editor, *31st International Conference on Principles and Practice of Constraint Programming, CP 2025, Glasgow, Scotland, August 10-15, 2025*, volume 340 of *LIPICs*, pages 40:1–40:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Appendix A

Data Generation with CLP Queries

Research Database	Query 1	Query 2
ACM Digital Library	Abstract:("data-driven testing" OR "data generation" OR "data creation" OR "input generation" OR "input creation" OR "generate data" OR "create data" OR "generate input" OR "create input") AND Abstract:("constraint programming" OR "program constraints" OR "constraint-based" OR "constraint logic programming")	Abstract:("test generation" OR "generate tests" OR "test case generation" OR "test creation" OR "create tests" OR "test case creation") AND Abstract:("constraint programming" OR "program constraints" OR "constraint-based" OR "constraint logic programming")
IEEE Xplore	("Abstract":"data-driven test*" OR "Abstract":"data" NEAR/1 "generat*" OR "Abstract":"data" NEAR/1 "creat*" OR "Abstract":"input" NEAR/1 "generat*" OR "Abstract":"input" NEAR/1 "creat*") AND ("Abstract":"constraint" NEAR/1 "program*" OR "Abstract":"constraint-based" OR "Abstract": "constraint logic program*")	("Abstract":"test*" NEAR/1 "generat*" OR "Abstract": "test* case* generat*" OR "Abstract":"test*" NEAR/1 "creat*" OR "Abstract": "test* case* creat*") AND ("Abstract":"constraint" NEAR/1 "program*" OR "Abstract":"constraint-based" OR "Abstract": "constraint logic program*")
Scopus	((ABS("data-driven test*") OR ABS("data" W/0 "generat*") OR ABS("data" W/0 "creat*") OR ABS("input" W/0 "generat*") OR ABS("input" W/0 "creat*")) AND (ABS("constraint" W/0 "program*") OR ABS("constraint-based") OR ABS("constraint logic program*")))	((ABS("test*" W/0 "generat*") OR ABS ("test* case* generat*") OR ABS("test* W/0 creat*") OR ABS("test* case* creat*")) AND (ABS("constraint" W/0 "program*") OR ABS("constraint-based") OR ABS("constraint logic program*")))

Table A.1: Queries on Data Generation with CLP