University of Naples Federico II

Polytechnic and Basic Sciences School

Department of Electrical Engineering and Information Technology

Master's Degree in Computer Science

Design and implementation of a model checking tool for verification of linear temporal properties on polyhedral systems

Thesis advisors
Prof. Massimo Benerecetti
Prof. Fabio Mogavero
Prof. Marco Faella

Thesis Co-Advisor
Prof. Alessandro De Luca

Candidate
Vincenzo Tramo
N97000433

Academic Year 2024–2025

University of Naples Federico II

Polytechnic and Basic Sciences School

Department of Electrical Engineering and Information Technology

Master's Degree in Computer Science

Design and implementation of a model checking tool for verification of linear temporal properties on polyhedral systems

Thesis advisors
Prof. Massimo Benerecetti
Prof. Fabio Mogavero
Prof. Marco Faella

Candidate
Vincenzo Tramo
N97000433

Thesis Co-Advisor
Prof. Alessandro De Luca

Academic Year 2024–2025

Abstract

This thesis presents the design and implementation of a model checking tool for verifying real-time temporal properties over continuous dynamical systems described by polyhedral differential inclusions. These systems, also known as single-location linear hybrid systems, evolve under constant, convex flow constraints and are particularly relevant for the formal analysis of cyber-physical systems. Leveraging recent results that establish the decidability of model checking for this class of systems, we develop a verification framework based on Real-Time Logic (RTL), a continuous-time interpretation of Linear Temporal Logic (LTL). The tool symbolically computes the set of all initial states from which the system satisfies a given temporal property, using automata-theoretic constructions and polyhedral operations. Implemented in C++, the tool integrates the Parma Polyhedra Library (PPL) for geometric reasoning and the SPOT library for automata generation. Multiple verification modes are supported, including finite-time and infinite-time reasoning. The implementation is validated through a series of experiments, which assess both correctness and performance. This work provides a practical, extensible foundation for the exact verification of real-time properties in continuous systems, contributing to the broader effort of bridging formal methods with real-world system analysis. The source code is publicly available at [26].

Contents

1 Introduction **1**

I Theoretical Background 7

2 Polyhedral Systems **9**
- 2.1 Polyhedra and Polytopes 9
  - 2.1.1 Patches 10
  - 2.1.2 Topological Properties 11
- 2.2 Atomic Propositions 16
  - 2.2.1 Observables 17
  - 2.2.2 Space Partition into Polyhedra 17
- 2.3 Definition of Polyhedral System 18
- 2.4 System Trajectories and Signals 19
  - 2.4.1 Trajectories 20
  - 2.4.2 Trajectory Termination Criteria 20
  - 2.4.3 Signals 22
  - 2.4.4 Time slicing 22
  - 2.4.5 Well-Behavedness and Finite Variability 22
- 2.5 Reachability Operators on Polyhedra 23
  - 2.5.1 Monotonicity Properties of Reach Operators 23
  - 2.5.2 Distributivity Properties of Reach Operators 24
  - 2.5.3 Computing the Reach Operators 24
  - 2.5.4 Split Function 25
  - 2.5.5 Computational Complexity of Reach Operators 26

3 Linear Temporal Logics **27**
- 3.1 Syntax and Operators 27
- 3.2 Linear Temporal Logic Semantics 28
- 3.3 Real-Time Logic Semantics 28
- 3.4 Non-Recurrent Fragment of `RTL` 29

4 **Solving the Model Checking Problem for $\mathtt{RTL}_\omega/\mathtt{RTL}_f$** 31
4.1 The Model-Checking Problem 31
4.1.1 Existential Denotation 31
4.1.2 Universal Denotation 32
4.2 Decidability Scenarios for each Semantics 32
4.3 Discretisation of an `RTL` Formula 33
4.3.1 Discretising Signals 34
4.3.2 Discretising Formulae 36
4.3.3 Automaton Construction for $\mathtt{RTL}_\mathtt{f}/\mathtt{RTL}_\omega$ 36
4.4 Finite-Time Semantics 37
4.4.1 General Case 38
4.4.2 Omnidirectional Flow Scenario 42
4.5 Infinite-Time Semantics 43
4.5.1 Omnidirectional Flow Scenario 43
4.5.2 Non-Recurrent $\mathtt{RTL}_\omega$ and Compact Flow Scenario 43
4.6 Maximal Semantics (may, must) 44
4.6.1 The *brink* Atomic Proposition 45
4.6.2 Omnidirectional Flow Scenario 46
4.6.3 Forced Motion and Bounded Invariant Scenario 46
4.6.4 Non-Recurrent `RTL` and Compact Flow Scenario 47

II **Design & Implementation** 49

5 **Requirements and Analysis of the `RTL` Model Checking Tool** 51
5.1 Functional and Non-Functional Requirements 51
5.1.1 Functional Requirements Specification 51
5.1.2 Non-Functional Requirements Specification 53
5.2 High-level Architecture and Analysis Class Diagrams 53
5.2.1 High-Level Architecture 54
5.2.2 Analysis Class Diagrams 55

6 **Project Structure and Fundamental Libraries** 63
6.1 Project Structure and Dependencies 63
6.1.1 The Meson Build System 64
6.1.2 Source Code Organisation 65
6.1.3 Source Modules and Their Roles 67
6.1.4 Project Dependencies 68
6.2 Parma Polyhedra Library (PPL) 69
6.2.1 Not Necessarily Closed Convex Polyhedron 70
6.2.2 Double Description Method for Polyhedra 70
6.2.3 Representing General Polyhedra 70
6.2.4 Type Aliases for PPL Structures (`src/utils/ppl/ppl_aliases.h`) 71

6.2.5 Main Operations on Polyhedra 72
6.2.6 Utility Functions for PPL (`src/utils/ppl/ppl_utils.h`) . . 73
6.3 Spot 75
6.3.1 Spot Architecture Overview 76
6.3.2 Constructing Formulae using `spot::formula` 76
6.3.3 Parsing LTL Formulae 78
6.3.4 Transition-based ω-Automata 79
6.3.5 Binary Decision Diagrams 80
6.3.6 Translation of LTL Formulae into Automata 81
6.3.7 Working with $\mathtt{LTL}_\mathrm{f}$ Formulae with Finite-Time Semantics . . . 81
6.4 ANTLR4 82
6.5 Grammars, Parsers, and Lexers 82
6.6 Parse Tree Walking Mechanisms: Listeners and Visitors 83

7 Polyhedral Systems: Specification Language and Representation 85
7.1 Polyhedral System Specification Language 85
7.1.1 Polyhedral System Grammar 86
7.1.2 Parse a Polyhedral System 87
7.1.3 Handling Parsing Errors 88
7.2 Polyhedral System Representation 90
7.2.1 Giving Interpretations to Atomic Propositions 90
7.2.2 Invariant Properties 92
7.2.3 Flow Properties 93
7.2.4 Computing the Pre Flow 95
7.2.5 Observable Generation 95
7.2.6 Extending a Polyhedral System 97
7.2.7 Preserving User-Defined Variable Names in Polyhedra String Representations 97
7.2.8 Constructing a Polyhedral System with `PolyhedralSystemBuilder` 101
7.2.9 Class Diagram of the `system` Component 102

8 Representing and Discretising `RTL` Formulae 103
8.1 `RTL` Specification Language 103
8.1.1 `RTL` Grammar 104
8.1.2 Parse an `RTL` Formula 104
8.1.3 `RTL` Parsing Result Representation 105
8.2 `RTL` Formula Utilities and Constants: the `formula` Component 106
8.2.1 Utilities and Operations on Formulae 107
8.2.2 `RTL` Predefined Constants and Special Formulae 107
8.2.3 Discretisation Transformation Rules 112
8.2.4 Optimising Discretisation Process for Finite-Time Semantics . 113

9 Automata Are All You Need 115
9.1 The Base Abstract Class for Automata 115
9.1.1 Interface Design and Core Operations 115
9.1.2 Acceptance Conditions 116
9.2 Polyhedral Automata for LTL 118
9.2.1 From Edge-Based Denotation to State-Based Denotation . . . 118
9.2.2 Computing State Denotations with Don't-Care Observables . 120
9.2.3 Preprocessing the Transition-Based Automaton 122
9.2.4 Postprocessing the State-Based Automaton 122
9.2.5 Concrete Polyhedral Automata: Büchi, NFA, and Backward NFA 122
9.3 Polyhedral Abstractions 123
9.3.1 Omnidirectional Polyhedral Abstraction 124
9.4 Polyhedral Synchronous Product Automaton 126
9.5 Automata Emptiness Check Algorithms 127
9.6 On-the-Fly Algorithm for Finite-Time Semantics 128

10 Solver Hierarchy 131
10.1 Solver Interface 131
10.2 Explicit Solvers 132
10.2.1 Omnidirectional Solvers 133
10.3 On-The-Fly Solvers 133
10.4 Solver Selection Flow Diagrams 135

III Experimental Framework and Results 139

11 Experimental Infrastructure and Tools 141
11.1 `RTL` Formulae Generation with `rtl-gen` 141
11.2 Polyhedral System Generation with `sys-gen` 142
11.2.1 GAP Experiment: Generalisation and Implementation 143
11.3 From Polyhedra to GeoGebra: The `poly-ggb` Tool 148
11.3.1 Usage and Options 149
11.3.2 Polyhedra Specification for `poly-ggb` 150
11.3.3 Example: Visualising an Instance of the GAP Experiment . . 150
11.4 Built-in Statistics Engine of `rtl-mc` 153

12 Model Checking Results and Performance Evaluation 157
12.1 GAP Experiment `RTL` Model Checking Results 157
12.1.1 Incremental GAP Thickness under Different Flow Constraints 159
12.1.2 Incremental Time 167
12.1.3 Visual Certificates of Existential Denotation Membership . . . 168
12.2 Scalability Experiments 170
12.3 `RTL` Automaton Visualisation 172

12.4 Profiling On-the-Fly Execution: Fine-Grained Statistics 174

IV Conclusion 177

13 Conclusions and Future Work 179

Appendix A Command-Line Interface Reference for `rtl-mc` 181
- A.1 Input Specification Options 181
  - A.1.1 Polyhedral System Input 181
  - A.1.2 `RTL` Formula Input 181
- A.2 Verification Semantics Options 182
  - A.2.1 Trajectory Semantics 182
  - A.2.2 Verification Modes: Existential or Universal 182
  - A.2.3 Model-Checking Problem 182
- A.3 Automaton Formula Translation Optimisations 182
- A.4 Output Format 183
  - A.4.1 Verbosity Levels 183
  - A.4.2 Statistics Engine 183
- A.5 Advanced Options 183
  - A.5.1 Exporting all Automata in DOT Format 183
  - A.5.2 Exporting On-The-Fly Algorithm Execution in DOT Format . 183
  - A.5.3 Direct Discretisation of `RTL` Formulae 183
  - A.5.4 Concurrent Execution Mode 184
- A.6 Special Environment Variables 184

Bibliography 185

–1–

Introduction

Ensuring system correctness against specifications has been a central pursuit in computer science for decades, addressed through the rigorous methods of formal verification. Among these techniques, model checking has risen to prominence as a particularly powerful and broadly adopted approach. A crucial and challenging application domain for these verification methods is that of hybrid systems. These represent a category of complex dynamical systems integrating both event-driven discrete logic and time-driven continuous evolution. These distinct dynamics interact, collectively determining the system's behaviour over time. Hybrid systems emerge naturally from the interaction between digital computation/communication and continuous physical processes, finding common application in areas like manufacturing, power grids, traffic management, and sensor networks. Motivated by both their inherent theoretical complexities and significant practical relevance, hybrid systems have been a major focus of research across control theory, mathematics, and computer science since the early 1990s. Today, the theory of hybrid dynamical systems is increasingly pivotal in shaping progress within cutting-edge domains like cyber-physical systems (CPS), autonomy, future transportation systems, and smart cities.

To formally capture the behaviour of hybrid systems, hybrid automata [1] provide a widely adopted mathematical model. Their effectiveness is demonstrated by successful applications in representing diverse complex systems, including gene regulatory networks [2], communication protocols [3], and robotic systems [4]. Central to the analysis of hybrid automata are two fundamental verification problems: the reachability problem, determining if a specific state or set of states can be reached, which is crucial for verifying safety properties (ensuring undesirable states are avoided) and the more general ω-language emptiness problem used to verify liveness properties. However, a significant hurdle arises: the decidability of these problems, whether an algorithm exists that can always provide a definite yes/no answer in finite time, depends critically on the specific class of hybrid automata under consideration [5]. Different approaches have been developed to solve verification problems for hybrid systems: restricting to special classes that admit exact verification, computing approximations of the reachable set, computing discrete abstractions of the system, simulating the system to find counterexamples, and using automated theorem proving techniques. For certain restricted classes with simpler dynamics, such as timed automata and initialised rectangular automata, the reachability problem is decidable. This allows, in principle, for the exact computation of reachable states and the application of verification techniques adapted from finite-state systems. Several well-established tools leverage this decidability. For instance, `Uppaal` [6], jointly developed by Uppsala University and Aalborg University using Java and C++, focuses specifically on timed automata. It provides a graphical user interface, supports the composition of automata, verifies properties expressed in a subset of CTL temporal logic, guarantees formally sound results, and demonstrates good scalability for its domain, handling systems with up to approximately 100 continuous clock variables. Another tool is `HyTech` [7], developed at Cornell and Berkeley by Henzinger, Ho, and Wong-Toi. Written in C++ with a textual scripting interface, `HyTech` was among the first tools capable of analysing hybrid systems, particularly rectangular automata. While operating on a decidable class, its computations were not always guaranteed to terminate, and it typically handled a smaller number of continuous variables (around 10). Unfortunately, this tractability does not extend to all useful classes of hybrid automata. Computing exact reachable states becomes computationally infeasible or even theoretically impossible for systems with sufficiently complex dynamics. In particular, for important classes such as linear hybrid automata (LHA), the reachability problem is known to be undecidable [5]. This fundamental limitation has spurred the development of approximate or incomplete methods. These techniques are implemented in tools like `SpaceEx` [8] and `NYCS` [9], offering practical means to analyse such complex systems despite the theoretical barriers. Developed at Verimag by a team including O. Maler and G. Frehse, `SpaceEx` is a C++ tool with a graphical user interface for analyzing hybrid automata with affine dynamics and guards ($\dot{x} = Ax + b$). It supports the composition of automata and computes over-approximations of the reachable set, representing the state space using support functions and polytopes. While it is recognised as one of the most scalable tools currently available, capable of handling systems with up to approximately one hundred continuous variables, its results are not guaranteed to be numerically sound. `NYCS` was developed at the University of Naples Federico II by M. Faella and M. Benerecetti. `NYCS` implements algorithms that often succeed in practice, providing a valuable analysis tool for verifying safety and reachability properties. The landscape of analysis tools for hybrid and related complex dynamical systems extends further. Notable examples include: `PHAVer` [10], one of the first efficient tools also originating from Verimag, focused on affine dynamics using polytope-based over-approximations, guaranteeing formal soundness but with scalability typically limited to around 10 variables. Tools tackling non-linear dynamics include `Flow`* [11] (RWTH Aachen, UC Boulder), which uses Taylor models for numerically sound over-approximation of polynomial systems, and `Ariadne` [12] (Maastricht, Verona, Padova), a C++ library providing sound over- and under-approximations for non-linear systems using techniques like Taylor image sets and kd-trees. Another library is `HyPro/HyDRA` [13] (RWTH Aachen) that offers a flexible C++ framework supporting various state-set representations (with numerical soundness depending on the chosen representation) and efficient algorithms for linear hybrid systems analysis. Differing slightly in focus, `SAPO` [14] (Verimag, Udine) is designed for reachability and parameter synthesis specifically for discrete-time polynomial systems, employing Bernstein polynomials and linear programming. This diverse ecosystem of tools highlights the variety of approaches developed to handle the challenges posed by different classes of hybrid and continuous systems.

Whereas reachability in linear hybrid systems (LHA) is undecidable, recent work by my thesis advisors [15] has demonstrated that model checking linear temporal logic (LTL) on a *single-location* is decidable under some assumptions. A *single-location system*, in this context, is governed by a single set of continuous dynamics without any discrete transitions that switch the governing dynamics. Unlike typical hybrid automata which feature multiple locations (or modes), each potentially with its own dynamics, and discrete transitions between them, a single-location system operates under a uniform set of continuous dynamics throughout its evolution. Therefore, while originating from the LHA framework, these systems are technically not "hybrid" in the standard sense, but rather fall into the category of *continuous-time* and *infinite-state* systems, which are crucial for cyber-physical applications [16], including embedded control systems, autonomous vehicles, and industrial automation. By restricting our analysis to systems without discrete location changes, we move from the generally undecidable landscape of LHA verification to a domain where LTL model checking is decidable, enabling the pursuit of exact verification algorithms. This thesis, therefore, investigates this specific, decidable subclass, exploring a setting where exact verification is feasible, distinct from the challenges posed by general hybrid systems. This positions our study within a particular niche, concentrating on a specific, decidable class of single-location systems. While their continuous dynamics are richer than those found in simpler decidable models like timed automata and initialized rectangular automata, they crucially lack the discrete switching between multiple modes—a characteristic feature and a primary source of undecidability in the general LHA framework. Our focus is thus on rigorously analyzing systems within this distinct category where exact verification remains feasible. Leveraging this theoretical foundation where exact verification is possible, the core contribution of this thesis is the practical realization of these concepts. Therefore, driven by the goal of providing exact algorithmic solutions within this decidable domain, the central focus of this thesis is the design and C++ implementation of a practical model-checking tool. This tool is dedicated to verifying LTL properties on single-location linear hybrid automata precisely, without resorting to approximation techniques.

We represent the state of these systems using a vector x of real-valued variables. The evolution of this state is governed by a differential inclusion of the form $\dot{x} \in F$. Here, $\dot{x}$ denotes the vector of derivatives of the state variables x. The set F, known as the *flow*, is a constant convex polyhedron, meaning it remains fixed and does not change over time. The constraint $\dot{x} \in F$ therefore means that, at any time instant, the velocity vector $\dot{x}$ must belong to this specific, time-invariant polyhedron F, which defines the set of permissible directions and rates of change for the system's state.

Since the systems under study evolve continuously over time, standard discrete-time interpretations of Linear Temporal Logic (LTL) are not directly applicable. We require a formalism capable of expressing temporal properties within a continuous-time context. For this purpose, we use RTL (Real-Time Logic) [17] to specify the properties to be verified. RTL provides a framework for interpreting LTL formulae over real-time or continuous system behaviours.

In this RTL interpretation, the connection between the logical formulae and the continuous system state is established through the atomic propositions (denoted, for example, by p). Each atomic proposition p is associated with a specific region in the state space, represented as a non-necessarily convex polyhedron. This polyhedron defines the set of states where the proposition p holds true. Importantly, these polyhedra do not necessarily have to be convex, allowing for the representation of complex state-space regions. The precise definition of which polyhedron corresponds to each atomic proposition is specified by the user, tailoring the verification to the properties of interest. Despite the real-time semantics, RTL preserves the standard syntax of LTL. Consequently, users can leverage the familiar LTL operators to construct temporal properties over the system's evolution, potentially including explicit time constraints.

The system's evolution is governed by the constant differential inclusion $\dot{x} \in F$, which effectively acts as a generator for the possible state trajectories. The set of trajectories mathematically satisfying this constraint may include those that exhibit infinitely rapid oscillations between distinct regions defined by atomic propositions. These phenomena, commonly referred to as *Zeno behaviour*, can be physically unrealistic and pose challenges for the symbolic analysis framework we aim to develop. Therefore, we impose the important assumption of *finite variability.* This assumption stipulates that, along any system trajectory, the truth value of any atomic proposition can change only a finite number of times within any bounded time interval. Given that our atomic propositions correspond to polyhedral regions in the state-space, finite variability means we focus exclusively on trajectories that cross the boundaries between these regions a finite number of times within any finite period. We refer to such trajectories as *well-behaved.*

We tackle the problem of verifying an RTL formula ϕ against a *polyhedral system*, which represents the unique mode in a single-location linear hybrid automaton. This polyhedral system consists of a flow constraint F that specifies the polyhedral inclusions governing the derivative of the system trajectories, as well as a polyhedral interpretation for all atomic propositions present in the formula. Given an RTL formula φ, a polyhedral system, and an initial point $p \in \mathbb{R}^n$, the model-checking problem asks whether there exists a trajectory starting from p that satisfies the formula φ. Instead of solving the model-checking problem for a specific initial point, we focus on computing a symbolic representation of the set of all initial points from which a trajectory satisfying the formula can emerge. This is referred to as the RTL *existential denotation problem.* The symbolic algorithm relies on translating an RTL formula into an LTL formula, followed by the standard automata construction for LTL [18, 19]. This automaton is then intersected with an appropriate finite-state abstraction of the polyhedral system. We identify several classes of trajectories and formulae for which the existential denotation problem is decidable, made possible through suitable discretisations of the polyhedral system. However, as described in previous work [15], when RTL is interpreted using finite semantics, an alternative, exact on-the-fly algorithm can be employed. This approach offers a potentially significant advantage as it avoids the explicit construction of the finite-state abstraction of the polyhedral system and, consequently, the computation of the intersection between this abstraction and the LTL automaton.

This thesis presents the design and implementation of an RTL model checking tool, developed in C++, which automates the verification process briefly described above. The tool enables users to specify a polyhedral system and an RTL formula to be checked, providing the flexibility to define the desired semantics of the formula for the verification process. At the core of the tool's design are two primary libraries: the Parma Polyhedra Library [20], a library for symbolic manipulation of polyhedra, and the SPOT library [21], a comprehensive framework for working with LTL formulae, ω-automata, and model checking. The complete source code of the tool is publicly available in a GitHub repository [26].

This thesis is structured into three parts. **Part I: Theoretical Background** lays the formal foundation for the verification framework developed in this work. Chapter 2 introduces *polyhedral systems*, beginning with the mathematical definitions of polyhedra and polytopes, and discusses their key topological properties such as closure, interior, and adjacency. The notion of system *trajectories* is formalised, along with several termination conditions that define four distinct semantics: finite-time, infinite-time, may, and must. The chapter also presents the concept of *signals*, and introduces the core *reachability operators* used to reason about possible system behaviours.

Chapter 3 introduces Linear Temporal Logic (LTL) and its continuous-time extension, Real-Time Logic (RTL), which is used to express temporal properties over real-valued signals. Both finite-time and infinite-time semantics are discussed. The chapter also identifies and characterises the *non-recurrent fragment* of RTL, a syntactic subset particularly well-suited for symbolic verification.

Chapter 4 formally defines the model-checking problems addressed in this work: verifying whether a polyhedral system satisfies a given RTL property under various semantics. The chapter introduces the concepts of existential and universal denotations and discusses decidability results for different system classes and formula fragments. It presents a symbolic discretisation technique that enables the reduction of continuous-time verification problems to discrete ones, allowing automata-based analysis. This leads to the construction of suitable automata and abstractions, and culminates in an on-the-fly algorithm for the finite-time semantics.

Part II: Design and Implementation focuses on the development of the verification tool. Chapter 5 defines the functional and non-functional requirements that guided the design, and presents the high-level architecture through component and class diagrams.

Chapter 6 details the internal structure of the tool, including its source code organisation, build system, and major external dependencies. Special attention is given to the two core libraries underpinning the tool's capabilities: the Parma Polyhedra Library (PPL) for symbolic geometric reasoning, and the SPOT library for automata-based temporal logic analysis.

Chapters 7 and 8 introduce the specification languages and internal representations for polyhedral systems and RTL formulae, respectively. The former focuses on parsing and representing polyhedral dynamics, while the latter discusses formula manipulation and discretisation.

Chapter 9 presents the automata-based machinery used to encode both temporal formulae and system abstractions. It discusses the design of automata, the encoding of RTL semantics, and the integration of geometric and temporal reasoning.

Chapter 10 describes the hierarchy of solver implementations integrated into the tool, each tailored to specific semantic interpretations and system structures. The chapter distinguishes between solvers based on explicit abstraction and those employing on-the-fly strategies, and discusses how solver selection depends on properties such as flow compactness and directionality.

Part III: Experimental Framework and Results provides empirical validation and performance analysis. Chapter 11 presents the experimental infrastructure and supporting tools developed to automate formula generation, polyhedral system instantiation, model-checking execution, and visualisation.

Chapter 12 reports a wide range of experiments. These include correctness checks across semantic variants, evaluation of denotation growth as a function of system parameters, and scalability studies with increasing formula complexity and system dimensionality. It also includes in-depth profiling of the on-the-fly algorithm, supported by the tool's built-in statistics engine.

The thesis concludes with Chapter 13, which summarises the main contributions and outlines future research directions, including opportunities for optimisation and theoretical extensions.

Part I

Theoretical Background

—2—

Polyhedral Systems

CONTENTS: **2.1 Polyhedra and Polytopes.** 2.1.1 Patches – 2.1.2 Topological Properties. **2.2 Atomic Propositions.** 2.2.1 Observables – 2.2.2 Space Partition into Polyhedra. **2.3 Definition of Polyhedral System.** **2.4 System Trajectories and Signals.** 2.4.1 Trajectories – 2.4.2 Trajectory Termination Criteria – 2.4.3 Signals – 2.4.4 Time slicing – 2.4.5 Well-Behavedness and Finite Variability. **2.5 Reachability Operators on Polyhedra.** 2.5.1 Monotonicity Properties of Reach Operators – 2.5.2 Distributivity Properties of Reach Operators – 2.5.3 Computing the Reach Operators – 2.5.4 Split Function – 2.5.5 Computational Complexity of Reach Operators.

The systems studied in this work are continuous-time dynamical systems, where the state vector $x \in \mathbb{R}^n$ evolves over time according to a differential inclusion of the form $\dot{x} \in \mathit{Flow}$. Here, Flow denotes a fixed, time-invariant convex polyhedron that constrains the possible directions and rates of change of the system state at any instant. This formulation naturally introduces non-determinism into the system's dynamics: at each point in time, the derivative $\dot{x}$ may lie anywhere within the polyhedron Flow. Since the evolution of the system is defined in terms of polyhedral constraints, and the properties we wish to verify are also interpreted over regions of the state space represented by polyhedra, a clear understanding of polyhedra and related geometric constructs is essential.

2.1 Polyhedra and Polytopes

A *convex polyhedron* is defined as the intersection of a finite number of half-spaces, which may be either strict or non-strict. A half-space is *strict* if it is defined by an inequality of the form $a_1x_1 + \cdots + a_nx_n < b$, and *non-strict* if the inequality is of the form $a_1x_1 + \cdots + a_nx_n \leq b$, where $a_1, \ldots, a_n \in \mathbb{R}$, $b \in \mathbb{R}$, and $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is the state vector. The boundary defined by a strict half-space is excluded from the polyhedron, whereas it is included in the case of a non-strict half-space.

An example of a simple convex polyhedron in $\mathbb{R}^2$ is shown in Figure 2.1. The region is defined as the intersection of five half-spaces: three with non-strict inequalities and two with strict ones. Geometrically, the resulting shape is a square located in the first quadrant. In this example, the boundary of a polyhedron consists of edges, which are straight line segments connecting two vertices. Two of the edges—the left and bottom sides—are excluded from the polyhedron due to the presence of strict inequalities. These open edges are depicted with dashed lines in the figure to visually indicate that their points do not belong to the polyhedron. The remaining three edges, defined by non-strict inequalities, are included in the region and are drawn as solid lines.

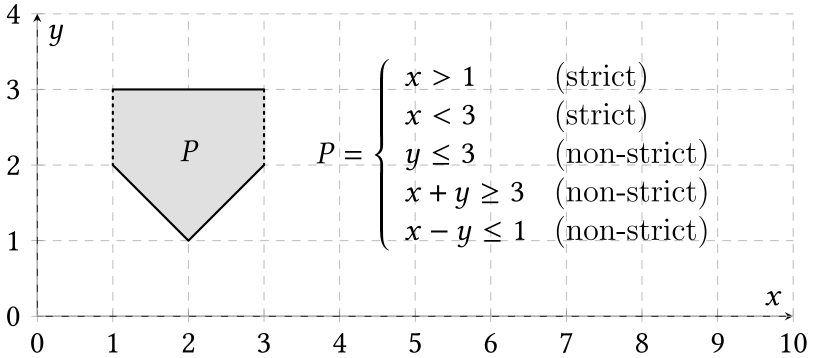


Figure 2.1: A convex polyhedron $P \subset \mathbb{R}^2$ defined as the intersection of five half-spaces. The dashed edges correspond to strict inequalities and are therefore not part of the polyhedron. The solid edges, corresponding to non-strict inequalities, are included in P.

A (general) *polyhedron* is a finite union of convex polyhedra, and a *polytope* is a convex polyhedron that is also bounded. We denote by $\mathsf{Poly}(\mathbb{R}^n)$ the set of all polyhedra in $\mathbb{R}^n$, and by $\mathsf{CPoly}(\mathbb{R}^n)$ the subset consisting of convex polyhedra. Throughout this thesis, we will typically use the letters P and Q to denote convex polyhedra, and the letters A and B for general (possibly non-convex) polyhedra.

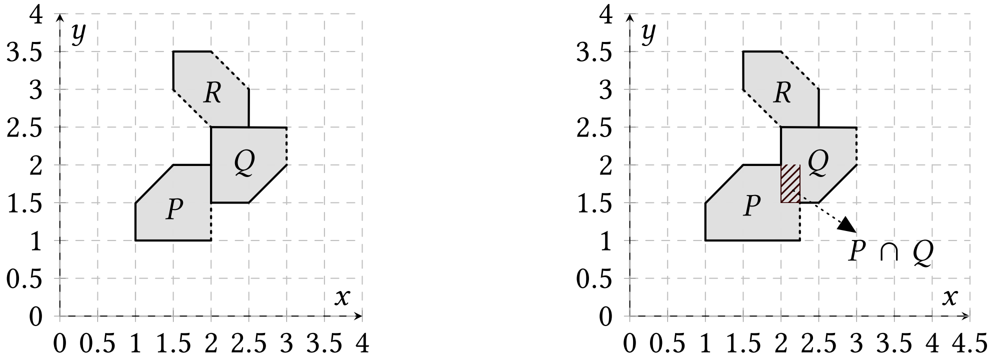
2.1.1 Patches

For a general polyhedron A, we denote by $\mathit{Patch}(A)$ a finite collection of convex polyhedra whose union constitutes A. The elements of this collection are called the *patches* of A, and provide a representation of A as a union of simpler convex components. While such a decomposition is not necessarily unique, it allows us to reason about general (possibly non-convex) polyhedra in terms of their convex parts.

Figure 2.2a illustrates a general polyhedron $A \subset \mathbb{R}^2$, composed of three convex patches labeled P, Q, and R. Each patch is a convex region, and their union forms the non-convex shape of A. In this example, the patch representation is explicitly given by:

$$\mathit{Patch}(A) = \{P, Q, R\}, \quad \text{with } P, Q, R \in \mathsf{CPoly}(\mathbb{R}^2).$$

This representation highlights how complex, non-convex regions can be effectively broken down into manageable convex components.



(a) A General Polyhedron with 3 Patches

(b) A General Polyhedron with 3 Non-Disjointed Patches

Figure 2.2: Representation of a General Polyhedra through Simple Convex Polyhedra

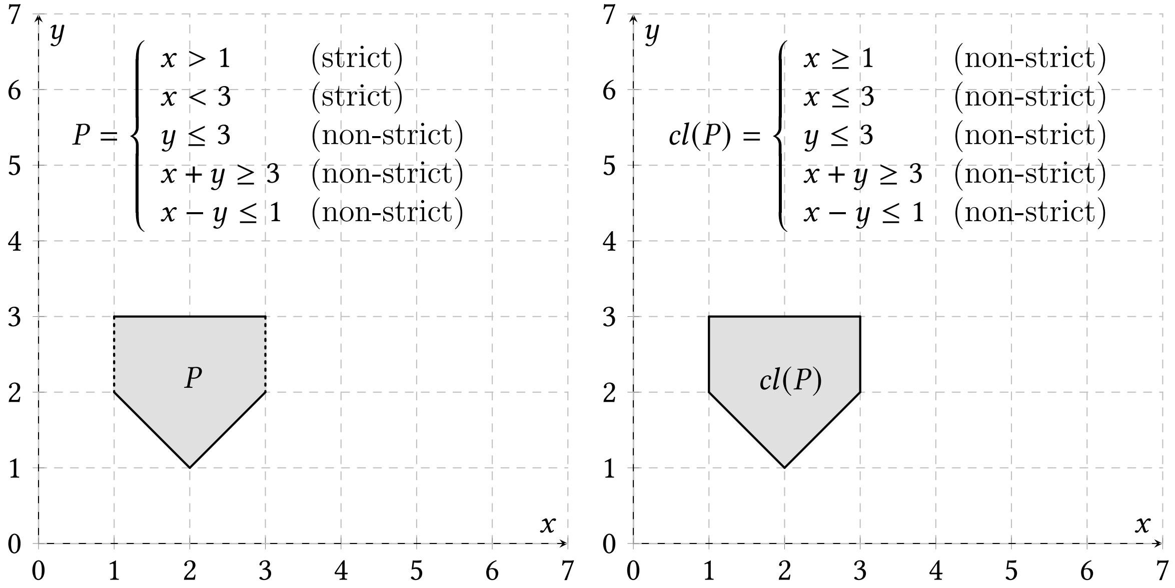
Note that the patches in a decomposition are not required to be disjoint. That is, a general polyhedron A may be represented by a collection of patches that partially overlap. Figure 2.2b illustrates such a case: it depicts a general polyhedron composed of three patches $P, Q, R \in \mathsf{CPoly}(\mathbb{R}^2)$, where $P \cap Q \neq \emptyset$. The intersection region between P and Q is highlighted in the figure, emphasizing that the overall shape of A is still the union of all patches, despite the fact that some of them share points. Such non-disjoint decompositions are perfectly valid and often arise naturally in symbolic model-checking algorithms, especially when patches are generated incrementally or through geometric operations such as intersections or preimages.

2.1.2 Topological Properties

Topological Closure

Given a convex polyhedron $P \in \mathsf{CPoly}(\mathbb{R}^n)$, we denote by $cl(P)$ its *topological closure*. Operationally, the closure is obtained by replacing all strict half-spaces in the definition of P with their non-strict counterparts. This has the effect of including all boundary points of P.

Figure 2.3 illustrates this concept. In subfigure 2.3a, the polyhedron P is defined using strict inequalities. In subfigure 2.3b, in the corresponding closure $cl(P)$, the strict inequalities have been replaced with a non-strict one. This example visually demonstrates how the topological closure operation expands the region to include its boundary.


$P = \begin{cases} x > 1 & \text{(strict)} \\ x < 3 & \text{(strict)} \\ y \le 3 & \text{(non-strict)} \\ x + y \ge 3 & \text{(non-strict)} \\ x - y \le 1 & \text{(non-strict)} \end{cases}$
P
$cl(P) = \begin{cases} x \ge 1 & \text{(non-strict)} \\ x \le 3 & \text{(non-strict)} \\ y \le 3 & \text{(non-strict)} \\ x + y \ge 3 & \text{(non-strict)} \\ x - y \le 1 & \text{(non-strict)} \end{cases}$
$cl(P)$
x
y
0
1
2
3
4
5
6
7

(a) A convex polyhedron P defined using strict constraints.

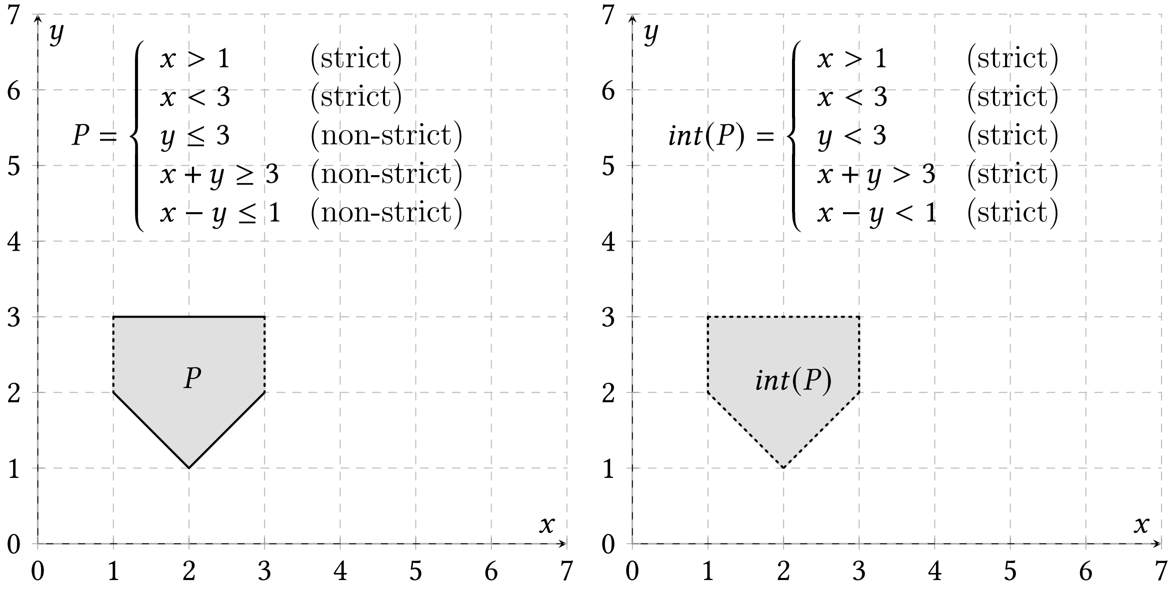
(b) The topological closure $cl(P)$, where the strict inequality has been replaced by a non-strict one.

Figure 2.3: Effect of the closure operation: boundary points excluded in the original polyhedron P (a) are included in its closure $cl(P)$ (b).

Interior

Dually, the *interior* of a polyhedron P, denoted $int(P)$, is obtained by replacing all inequalities in the definition of P with their strict versions. This operation removes the boundary of P, retaining only those points that lie inside the polyhedron P. Note that we always have $int(P) \subseteq P \subseteq cl(P)$.

Figure 2.4 illustrates this operation. Subfigure 2.4a shows a convex polyhedron P originally defined using non-strict inequalities. In subfigure 2.4b, all constraints have been replaced by strict ones, yielding the interior $int(P)$. The difference between the two highlights how the interior excludes boundary points.


$P = \begin{cases} x > 1 & \text{(strict)} \\ x < 3 & \text{(strict)} \\ y \leq 3 & \text{(non-strict)} \\ x + y \geq 3 & \text{(non-strict)} \\ x - y \leq 1 & \text{(non-strict)} \end{cases}$
P
x
y
0 1 2 3 4 5 6 7
$int(P) = \begin{cases} x > 1 & \text{(strict)} \\ x < 3 & \text{(strict)} \\ y < 3 & \text{(strict)} \\ x + y > 3 & \text{(strict)} \\ x - y < 1 & \text{(strict)} \end{cases}$
$int(P)$
x
y
0 1 2 3 4 5 6 7

(a) The polyhedron P defined with non-strict inequalities.

(b) The interior $int(P)$, obtained by converting non-strict inequalities to strict.

Figure 2.4: Comparison between a polyhedron defined with non-strict constraints (a) and its interior (b).

Adjacency and Border

Let $x \in \mathbb{R}^n$ and $P \in \mathsf{CPoly}(\mathbb{R}^n)$. We say that x is *adjacent* to P if $x \in cl(P)$; that is, x either belongs to P or lies on its boundary.

Figure 2.5 illustrates different cases of adjacency with respect to a convex polyhedron P, using four labeled points:

- Point p_1 lies in the interior of P, and is trivially adjacent, since $p_1 \in P$.
- Point p_2 lies on a edge of P that is defined by a non-strict inequality, and is therefore part of P. Thus, $p_2 \in P$ and is adjacent to P.

- Point p_3 lies exactly on a boundary that is excluded from P (defined by a strict inequality). While $p_3 \notin P$, we have $p_3 \in cl(P)$, so p_3 is still considered adjacent to P.

- Point p_4 lies strictly outside the region defined by $cl(P)$, and is therefore not adjacent to P.

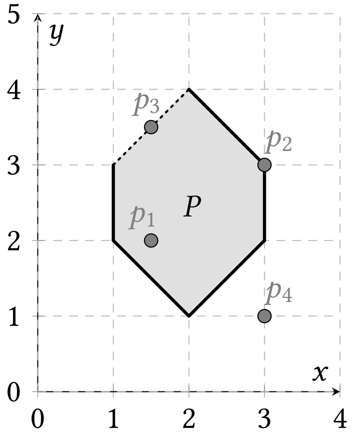


Figure 2.5: Illustration of point-to-polyhedron adjacency. Points p_1, p_2, and p_3 are adjacent to the convex polyhedron P, either because they lie within P or on its (possibly open) boundary. Point p_4 is not adjacent, as it lies outside the closure of P.

Two polyhedra $P, Q \in \mathsf{CPoly}(\mathbb{R}^n)$ are said to be *adjacent* if there exists at least one point that is adjacent to both, i.e., a point that belongs to the closure of both P and Q, and is contained in at least one of them. Formally, the *border* between P and Q is defined as:

$$border(P, Q) = (P \cap cl(Q)) \cup (cl(P) \cap Q).$$

If $border(P, Q) \neq \emptyset$, then P and Q are adjacent.

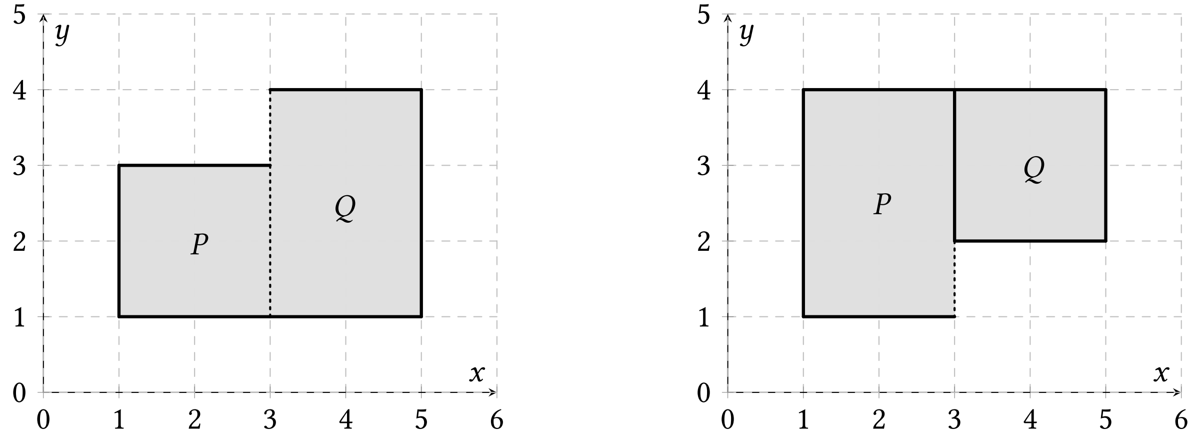
Figure 2.6 illustrates several configurations using different pairs of convex polyhedra P and Q:

- In subfigure 2.6a, P and Q are *not adjacent*. Although their boundaries geometrically coincide along an edge, both polyhedra define that edge using strict inequalities, so the shared boundary is not included in either closure.

- In subfigure 2.6b, P and Q are adjacent because $cl(P) \cap Q \neq \emptyset$.

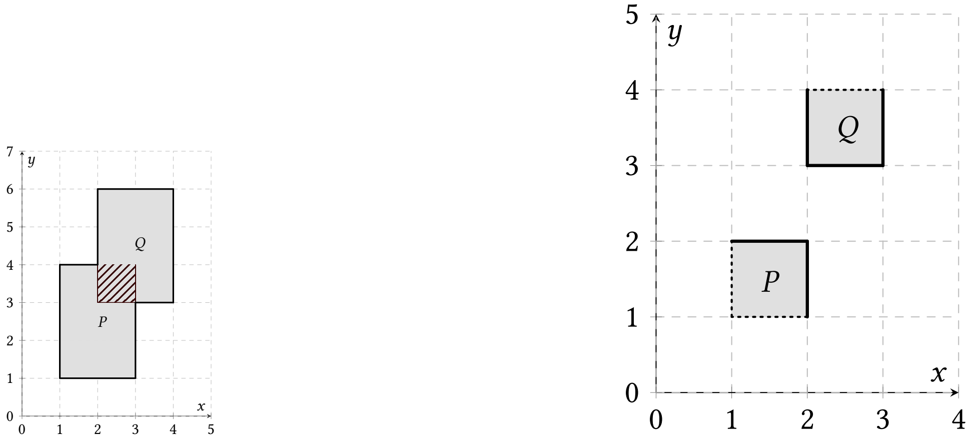
- In subfigure 2.6c, P and Q overlap, sharing interior points. This is a stronger form of adjacency, where the intersection $P \cap Q$ is non-empty.

- In subfigure 2.6d, the polyhedra are disjoint and distant from each other, and therefore not adjacent.



(a) Non-adjacent: the shared boundary edge is open in both P and Q.

(b) Adjacent: the edge of Q intersects the closure of P.



(c) Adjacent: P and Q share interior points.

(d) Non-adjacent: P and Q are disjoint and separated.

Figure 2.6: Examples of adjacency and non-adjacency between two convex polyhedra P and Q, based on the definition of their border.

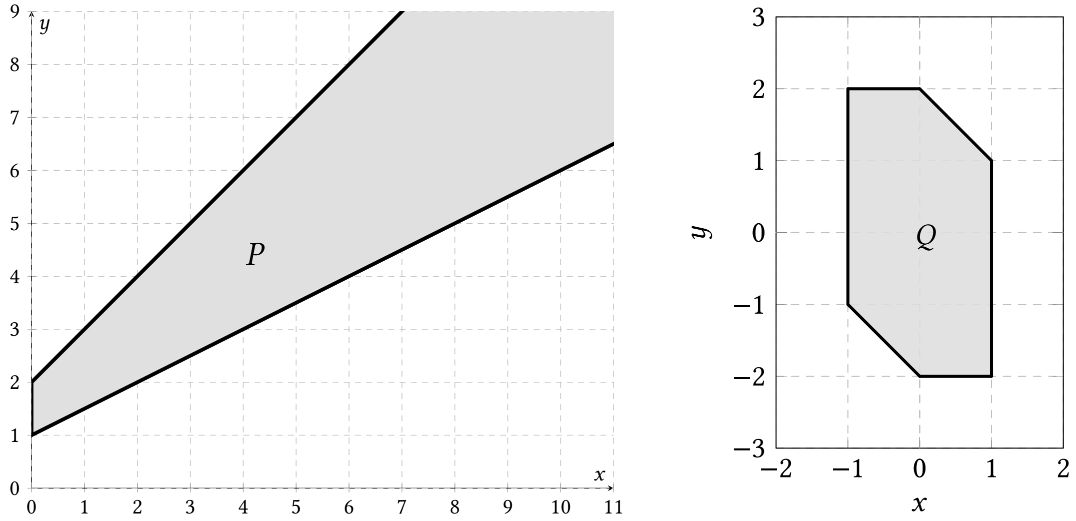
Compact Polyhedra

A polyhedron $P \subseteq \mathbb{R}^n$ is said to be *compact* if it is both *closed* and *bounded.*

A polyhedron is *closed* if it contains all of its boundary points; formally, this means that $P = cl(P)$. From a syntactic perspective, this condition is satisfied when all inequalities defining P are non-strict.

A polyhedron is said to be *bounded* if it does not extend infinitely in any direction. Geometrically, this means that the polyhedron has finite extent: it fits entirely within a sufficiently large ball in $\mathbb{R}^n$.

Figure 2.7 illustrates the difference between bounded and unbounded polyhedra. In subfigure 2.7a, the polyhedron is closed (all inequalities are non-strict) but unbounded: the region extends infinitely upwards and to the right. Since it is not bounded, it is not compact. In contrast, subfigure 2.7b shows a polyhedron that is both closed and bounded: it is entirely contained within a finite region of the plane and includes all its boundary points. Therefore, this second polyhedron is compact.



(a) A closed but unbounded polyhedron. The region extends infinitely in some directions.

(b) A closed and bounded polyhedron. The region is compact.

Figure 2.7: Comparison between a closed unbounded polyhedron (a) and a compact polyhedron (b).

2.2 Atomic Propositions

Let AP be a fixed, finite set of *atomic propositions.* Each atomic proposition $p \in \mathsf{AP}$ is interpreted as a subset of the state space $\mathbb{R}^n$, representing the set of points where the proposition holds true.

Formally, we assume the existence of a mapping

$$[\cdot] : \mathsf{AP} \to \mathsf{Poly}(\mathbb{R}^n),$$

which assigns to each atomic proposition $p \in \mathsf{AP}$ a corresponding polyhedron $[p] \in \mathsf{Poly}(\mathbb{R}^n)$, called its *interpretation*. The interpretation $[p]$ denotes the region of the state space where the proposition p holds. The notation $x \models p$ is used to indicate that the point $x \in \mathbb{R}^n$ satisfies the proposition p, meaning:

$$x \models p \quad \text{iff} \quad x \in [p].$$

2.2.1 Observables

Given a set of atomic propositions $\alpha \subseteq \mathsf{AP}$ (a.k.a. *observable*), we define the interpretation of α, denoted $[\![\alpha]\!]$, as the set of points in $\mathbb{R}^n$ where all and only the propositions in α hold. Formally,

$$[\![\alpha]\!] = \left(\bigcap_{p \in \alpha} [p]\right) \cap \left(\bigcap_{p \in \mathsf{AP} \setminus \alpha} \overline{[p]}\right).$$

By definition, $[\![\alpha]\!]$ is a polyhedron, as it results from finite intersections of polyhedra.

Note that $[\![\{p\}]\!] \subseteq [p]$, and this inclusion may be strict, since $[p]$ may also include points where other propositions hold.

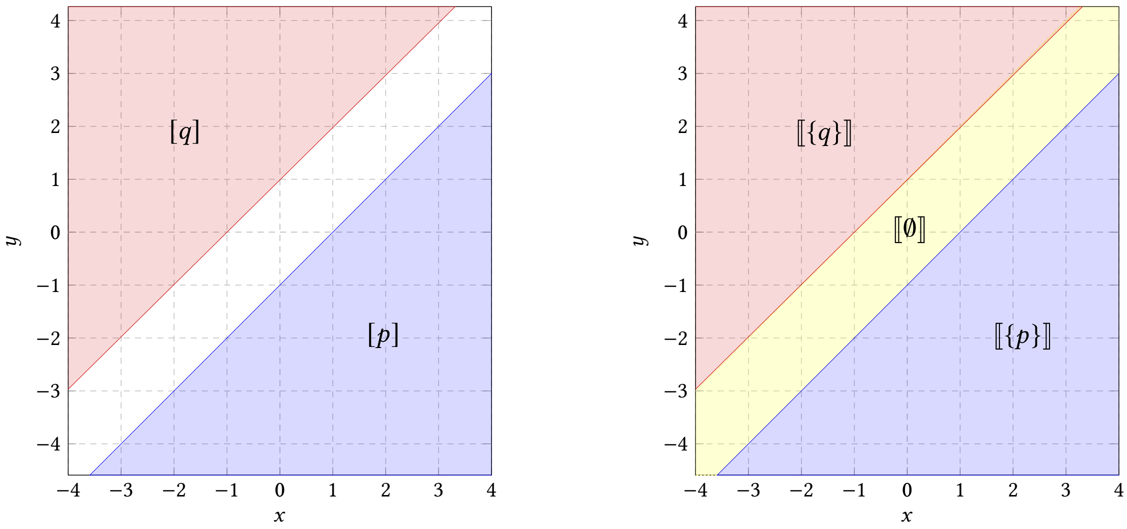
2.2.2 Space Partition into Polyhedra

The mapping $[\![\cdot]\!] : 2^{\mathsf{AP}} \to \mathsf{Poly}(\mathbb{R}^n)$ partitions the state space: for any $\alpha_1, \alpha_2 \subseteq \mathsf{AP}$, we have either $[\![\alpha_1]\!] = [\![\alpha_2]\!]$ or $[\![\alpha_1]\!] \cap [\![\alpha_2]\!] = \emptyset$. Therefore, the image of 2^{AP} under $[\![\cdot]\!]$ defines a finite partition of $\mathbb{R}^n$ into mutually disjoint polyhedral regions, each corresponding to a unique observable combination of propositions. This partition plays a central role in the abstraction of continuous systems: each point in the state space is associated with a unique observable $\alpha \subseteq \mathsf{AP}$.

Let us consider a basic example with two atomic propositions: $\mathsf{AP} = \{p, q\}$. We define their interpretations as follows:

$$[p] = \{(x, y) \in \mathbb{R}^2 \mid x \geq y + 1\}, \qquad [q] = \{(x, y) \in \mathbb{R}^2 \mid y \geq x + 1\}.$$

The region $[p]$ consists of the half-plane above the line $x = y+1$, including the boundary. The region $[q]$ consists of the half-plane below the line $y = x + 1$, also including the boundary. These two regions are disjoint and do not overlap: $[p] \cap [q] = \emptyset$ (see Figure 2.8a).



(a) The interpretation of atomic propositions p and q.

(b) Partition of the space into observables $[\![\alpha]\!]$.

Figure 2.8: Visualization of atomic propositions (a) and the induced partition (b).

Given $\mathsf{AP} = \{p, q\}$, there are four subsets $\alpha \subseteq \mathsf{AP}$, and for each we compute the observable region $[\![\alpha]\!]$ as defined earlier:

- $[\![\emptyset]\!] = \overline{[p]} \cap \overline{[q]}$: the region where neither p nor q hold — i.e., the strip between the two half-planes.
- $[\![\{p\}]\!] = [p] \cap \overline{[q]} = [p]$: since $[p] \cap [q] = \emptyset$, removing $[q]$ has no effect.
- $[\![\{q\}]\!] = [q] \cap \overline{[p]} = [q]$: analogous to the case above.
- $[\![\{p, q\}]\!] = [p] \cap [q] = \emptyset$: no point satisfies both.

As a result, the observable partition consists of three non-empty, disjoint polyhedral regions, and one empty region (see Figure 2.8b). This illustrates how a finite set of atomic propositions induces a finite partition of the continuous state space, where each region is labeled by the subset $\alpha \subseteq \mathsf{AP}$ of propositions that hold there.

2.3 Definition of Polyhedral System

We consider dynamical systems constrained by a fixed convex polyhedron $\mathit{Flow} \subseteq \mathbb{R}^n$ called *flow constraint*, which defines a polyhedral differential inclusion.

A *polyhedral system* is a triple $\mathcal{P} = (\mathit{Flow}, \mathit{Inv}, [\cdot])$, where:

- $\mathit{Flow} \subseteq \mathbb{R}^n$ is a fixed convex polyhedron representing the admissible directions of motion (the flow constraint);

- $Inv \subseteq \mathbb{R}^n$ is a closed polyhedron called the *invariant*;
- $[\cdot] : \mathsf{AP} \to \mathsf{Poly}(\mathbb{R}^n)$ is an interpretation function that assigns to each atomic proposition $p \in \mathsf{AP}$ a polyhedral subset $[p] \subseteq \mathbb{R}^n$.

The convex polyhedron $Flow$ is fixed and typically omitted from the notation when it is clear from the context. The closed polyhedron Inv constrains the evolution of the system: any trajectory must remain within this region for all time. The interpretation $[\cdot]$ provides the semantic meaning of atomic propositions, enabling the labeling of points in the continuous state space with sets of propositions that hold true at those points.

Figure 2.9 illustrates an instance of the flow constraint $Flow$, defined as $Flow = \{(x, y) \in \mathbb{R}^2 \mid 2 \leq x \leq 4 \wedge 2 \leq y \leq 4\}$. A point $p = (-4, 4)$ lies outside this region, and a cone originating from p represents the possible directions in which the system can evolve from that point, in accordance with the flow constraint $Flow$.

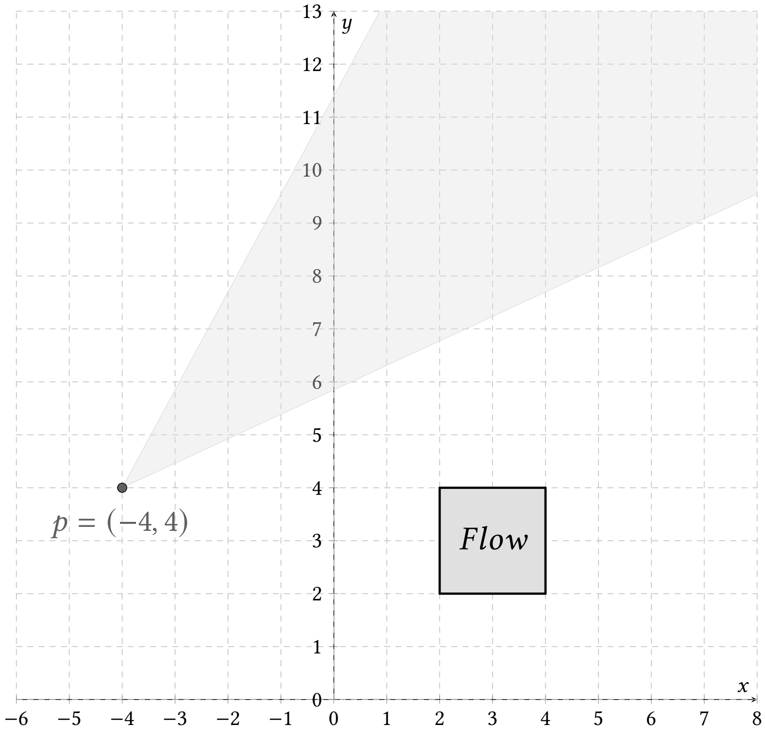


Figure 2.9: A flow constraint $Flow$ and a cone of directions from a point p

2.4 System Trajectories and Signals

In this section, we formalise the notions of system trajectories and signals over time. We first introduce some basic notations for time intervals, which will be used throughout the section.

For real values $a, b \in \mathbb{R}_{\geq 0}$, we use the notation $\langle a, b]$ to denote either of the two right-closed intervals: $(a, b]$ or $[a, b]$. Given an interval $I = \langle a, b]$, we define the suffix intervals $I_{>t} = (t, b]$ and $I_{\geq t} = [t, b]$, as well as the prefix intervals $I_{<t} = \langle a, t)$ and $I_{\leq t} = \langle a, t]$. Let Int denote the set of all right-closed intervals, including both bounded intervals where $\sup(I) \in I$ and unbounded intervals with $\sup(I) = \infty$.

2.4.1 Trajectories

Let $x \in \mathbb{R}^n$ be an initial point and $I \in \mathit{Int}$ a right-closed time interval. A *trajectory* from x with time domain I is a function $f : I \longrightarrow \mathbb{R}^n$ satisfying the following conditions:

1. $\lim_{t \to \inf(I)} f(t) = x$;
2. $f(t) \in \mathit{Inv}$ for all $t \in I$;
3. f is continuous;
4. whenever $\dot{f}(t)$ is defined, it holds that $\dot{f}(t) \in \mathit{Flow}$;
5. f is differentiable except at finitely many points in every bounded subinterval of I.

A trajectory is said to be *finite-time* if its time domain is bounded, and *infinite-time* otherwise. We use I_f to denote the time domain of a trajectory f. A trajectory f is *admissible* in the polyhedral system $\mathcal{P}$ if it remains within the invariant set for its entire duration, that is, $f(t) \in \mathit{Inv}$ for all $t \in I_f$. Given a point $x \in \mathbb{R}^n$, we write $\mathit{Traj}(x)$ for the set of all admissible trajectories starting at x.

Given two trajectories $f, f' \in \mathit{Traj}(x)$, we say that f' is a *suffix* (resp., *prefix*) of f if it coincides with the restriction of f to a suffix (resp., prefix) of its time domain—i.e., $I_{f'} = I_f \sim t$, for some $t \in I_f$ and $\sim \in \{>, \geq\}$ (resp., $\sim \in \{<, \leq\}$).

A trajectory is said to be *well-behaved* if, in any bounded interval $\langle a, b] \subseteq I_f$, it crosses every hyperplane only finitely many times. Formally, for any hyperplane H, there exists a finite sequence of time instants $a = t_0 < t_1 < \cdots < t_k = b$ such that, in each open subinterval (t_i, t_{i+1}), the trajectory lies in the same closed half-space defined by H. Further properties of well-behaved trajectories are discussed in Section 2.4.5.

We restrict $\mathit{Traj}(x)$ to denote the set of all well-behaved admissible trajectories starting from x.

2.4.2 Trajectory Termination Criteria

In the following, we adopt four different semantics for polyhedral systems, depending on two key aspects: the type of trajectories being considered—finite or infinite—and on the interpretation of the invariant. We begin with an intuitive overview of each semantics before giving their formal definitions.

[fin] **Finite-time semantics.** This semantics considers only *finite-time* trajectories. It is appropriate for reasoning about properties that can be positively verified on finite prefixes, such as *reachability* properties.

[inf] **Infinite-time semantics.** This semantics restricts attention to *infinite-time* trajectories, and is suitable for verifying *non-terminating* properties, such as *liveness* or *fairness*, where the system is expected to keep evolving indefinitely.

[may] **May-exit semantics.** A trajectory may either be infinite or terminate when reaching a *may-exit point*—a point on the boundary of the invariant from which at least one admissible direction leads outside the invariant. This semantics considers the invariant as the region of the state-space where the current dynamics hold. When reaching a may-exit point, the current dynamics may end because the system moves to another dynamics (not modelled in the current polyhedral system). That may be the case if we are verifying a local property of a single mode within a broader hybrid system.

[must] **Must-exit semantics.** A trajectory is either infinite or terminates at a *must-exit point*—a point on the boundary of the invariant from which *all* admissible directions exit the invariant. This is a stronger interpretation of the invariant as an inescapable constraint on motion. Under this interpretation, a trajectory only ends when no dynamics are possible within the invariant.

These four semantics induce four corresponding sets of well-behaved trajectories starting from a given initial point $x \in \mathbb{R}^n$, denoted respectively by $\mathit{Traj}^{\mathrm{fin}}(x)$, $\mathit{Traj}^{\mathrm{inf}}(x)$, $\mathit{Traj}^{\mathrm{may}}(x)$, and $\mathit{Traj}^{\mathrm{must}}(x)$.

- $\mathit{Traj}^{\mathrm{fin}}(x)$ contains the admissible trajectories of $\mathcal{P}$ with bounded domain.
- $\mathit{Traj}^{\mathrm{inf}}(x)$ contains the admissible trajectories with unbounded domain.

To formalize $\mathit{Traj}^{\mathrm{may}}(x)$ and $\mathit{Traj}^{\mathrm{must}}(x)$, we use the following notion. Given a polyhedron P and a trajectory f, we say that f *immediately enters* P if there exists a time $t \in I_f \setminus \{\inf(I_f)\}$ such that $f(t') \in P$, for all $\inf(I_f) < t' \leq t$.

Let $\mathit{Traj}_{\top}(x)$ denote the set of admissible trajectories from x in the polyhedral system obtained from the original one by replacing the invariant with the trivial constraint (i.e., the whole space $\mathbb{R}^n$).

Then, the may-exit and must-exit semantics are captured precisely as:

$$\mathit{Traj}^{\mathrm{may}}(x) = \mathit{Traj}^{\mathrm{inf}}(x) \cup \left\{ f \in \mathit{Traj}^{\mathrm{fin}}(x) \;\middle|\; \begin{array}{l} \exists f' \in \mathit{Traj}_{\top}(f(\sup(I_f))). \\ f' \text{ immediately enters } \overline{\mathit{Inv}} \end{array} \right\},$$

$$\mathit{Traj}^{\mathrm{must}}(x) = \mathit{Traj}^{\mathrm{inf}}(x) \cup \left\{ f \in \mathit{Traj}^{\mathrm{fin}}(x) \;\middle|\; \begin{array}{l} \forall f' \in \mathit{Traj}_{\top}(f(\sup(I_f))). \\ f' \text{ immediately enters } \overline{\mathit{Inv}} \end{array} \right\}$$

The following strict inclusions hold in general, and no others:

$$Traj^{\mathrm{inf}}(x) \subseteq Traj^{\mathrm{must}}(x)$$
$$Traj^{\mathrm{fin}}(x) \subseteq Traj^{\mathrm{must}}(x) \subseteq Traj^{\mathrm{may}}(x) \subseteq Traj(x)$$

2.4.3 Signals

The interpretation $[\cdot]$ of atomic propositions induces a mapping from trajectories to functions of type $I \longrightarrow 2^{AP}$, referred to as *signals* [22]. These signals form the semantic basis of the temporal logics introduced in Chapter 3. Given a trajectory f, we denote by σ_f the signal that, for each time instant $t \in I$, assigns the set of atomic propositions that hold at $f(t)$. Formally:

$$\sigma_f(t) \triangleq \{p \in AP \mid f(t) \in [p]\}.$$

Standard notions such as prefix, suffix, and the distinction between finite-time and infinite-time naturally extend from trajectories to signals.

2.4.4 Time slicing

A *time slicing* τ for an infinite-time signal $\sigma : I \longrightarrow 2^{AP}$ is an enumerable, increasing sequence of time points $\{t_i\}_{i=0}^{\omega} \subseteq \mathbb{R}^+$ such that $t_0 = \mathbf{inf}(I)$ and $\mathbf{sup}\{t_i\}_{i=0}^{\omega} = \mathbf{sup}(I)$. This sequence is associated with a corresponding sequence of observables $\{\alpha_i\}_{i=0}^{\omega} \subseteq 2^{AP}$ satisfying the condition that, for every index $0 \leq i < \omega$ and every time instant $t \in (t_i, t_{i+1})$, it holds that $\sigma(t) = \alpha_i$. The definition extends naturally to finite-time signals, where the slicing $\{t_i\}_{i=0}^{k} \subseteq \mathbb{R}^+$ is finite, but follows the same principle. A time slicing must include a time point every time the original signal moves from one observable to another. However, it may also include additional time points even when the observable remains constant—a phenomenon known as *stuttering*. We denote by $TS(\sigma)$ the set of all valid time slicings for a signal σ, and define $TS(f) \triangleq TS(\sigma_f)$ for any trajectory f. Importantly, the set $TS(\sigma)$ does not depend on whether the domain I is left-open or not; that is, $TS(\sigma) = TS(\sigma_{>\mathrm{inf}(I)})$.

2.4.5 Well-Behavedness and Finite Variability

When considering membership in a half-space as an observable, the well-behavedness condition introduced earlier imposes that the truth value of such an observable along a trajectory f can change only a finite number of times within any bounded time interval. This property ensures regularity in the evolution of observables over time. A signal $\sigma : I \longrightarrow 2^{AP}$ is said to have *finite variability* if, in every bounded time interval $\langle a, b\rangle \subseteq I$, it changes value only a finite number of times. Formally, this means that every finite-time sub-signal $(\sigma_{\geq a})_{\leq b}$ admits a time slicing. It follows directly that if a trajectory f is well-behaved, then the associated signal σ_f has finite variability.

In the remainder of this thesis, we shall leave the polyhedral system $\mathcal{P}$ implicit and focus exclusively on well-behaved trajectories. This implies that all considered signals exhibit finite variability.

Table 2.1: Notation used in the thesis: three types of trace-like objects (from the most concrete to the most abstract) and the time decomposition of a signal.

Type	**Name**	**Role**	**Symbol**
$I \to \mathbb{R}^n$	Trajectory	Behaviour of a polyhedral system	f
$I \to 2^{AP}$	Signal	Interpretation of RTL	σ
$\{0, 1, \ldots, k\} \to 2^{AP}$	Trace	Interpretation of LTL	w
$\{0, 1, \ldots, k\} \to \mathbb{R}^+$	Time slicing	Time decomposition of a signal	τ

2.5 Reachability Operators on Polyhedra

As a fundamental tool for addressing the model-checking problem studied in this work, we introduce a geometric reachability operator defined over polyhedral sets. This operator, denoted by $\mathit{reach}^\flat(A, B)$, takes as input two (possibly non-convex) polyhedra A and B, and returns the set of points in A that can reach B by following a continuous trajectory that remains entirely within the union $A \cup B$. The superscript $\flat$ indicates the nature of the timing constraint applied during the reachability analysis. Specifically:

- $\mathit{reach}^0(A, B)$ identifies the points in A that can enter B *immediately*.
- $\mathit{reach}^+(A, B)$ identifies the points in A that can enter B only after a *strictly positive delay*.

Formally, the two variants of the reachability operator are defined as follows:

$$\mathit{reach}^0(A, B) \triangleq \{x \in A \mid \exists f \in \mathit{Traj}(x), t > 0.\, \forall t' \in (0, t].\ f(t') \in B\},$$
$$\mathit{reach}^+(A, B) \triangleq \{x \in A \mid \exists f \in \mathit{Traj}(x), t > 0.\, f(t) \in B \text{ and } \forall t' \in (0, t).\ f(t') \in A\}.$$

2.5.1 Monotonicity Properties of Reach Operators

The reachability operators $\boldsymbol{reach}^0$ and $\boldsymbol{reach}^+$ are *monotonic* with respect to set inclusion in both their arguments, under certain convexity assumptions. Specifically:

- **Monotonicity in the first argument:** Let $A \subseteq C$, where A and C are (possibly non-convex) polyhedra, and let P be a convex polyhedron. Then:

$$\boldsymbol{reach}^0(A, P) \subseteq \boldsymbol{reach}^0(C, P) \quad \text{and} \quad \boldsymbol{reach}^+(A, P) \subseteq \boldsymbol{reach}^+(C, P)$$

- **Monotonicity in the second argument:** Let $P \subseteq Q$, where P and Q are convex polyhedra, and let A be a (possibly non-convex) polyhedron. Then:

$$\boldsymbol{reach}^0(A, P) \subseteq \boldsymbol{reach}^0(A, Q) \quad \text{and} \quad \boldsymbol{reach}^+(A, P) \subseteq \boldsymbol{reach}^+(A, Q)$$

2.5.2 Distributivity Properties of Reach Operators

The reachability operators reach^0 and reach^+ exhibit several useful distributivity properties with respect to set union. These properties are listed below.

- **Distributivity of reach^0 over union in the first argument:**

$$\mathit{reach}^0(A \cup B, C) = \mathit{reach}^0(A, C) \cup \mathit{reach}^0(B, C)$$

- **Distributivity of reach^0 over union in the second argument:**

$$\mathit{reach}^0(A, B \cup C) = \mathit{reach}^0(A, B) \cup \mathit{reach}^0(A, C)$$

- **Distributivity of reach^+ over union in the second argument:**

$$\mathit{reach}^+(A, B \cup C) = \mathit{reach}^+(A, B) \cup \mathit{reach}^+(A, C)$$

- **Subdistributivity of reach^+ over union in the first argument:**

$$\mathit{reach}^+(A, C) \cup \mathit{reach}^+(B, C) \subseteq \mathit{reach}^+(A \cup B, C)$$

2.5.3 Computing the Reach Operators

We now describe how to compute the values of the reachability operators reach^0 and reach^+ using a finite number of geometric operations, assuming that the second argument is a convex polyhedron. If the second argument is non-convex, the distributivity properties discussed in Section 2.5.2 can be exploited to reduce the computation to the convex case.

To this end, we first define the *positive pre-flow* $P_{\swarrow_{>0}}$ of a convex polyhedron $P \subseteq \mathbb{R}^n$. Intuitively, this is the set of points that can reach P after a strictly positive delay, following some direction in the flow constraint Flow. Formally:

$$P_{\swarrow_{>0}} \coloneqq \{x \in \mathbb{R}^n \mid \exists d \in \mathit{Flow}, t > 0.\, x + d \cdot t \in P\}.$$

Figure 2.10 illustrates this concept: a point p belongs to the positive pre-flow of P if there exists a direction allowed by the flow constraint along which p can reach P after a positive amount of time.

We now describe how to compute the values of the reachability operators reach^0 and reach^+ using a finite number of geometric operations, assuming that the second argument is a convex polyhedron. If the second argument is non-convex, the distributivity properties discussed in Section 2.5.2 can be exploited to reduce the computation to the convex case.

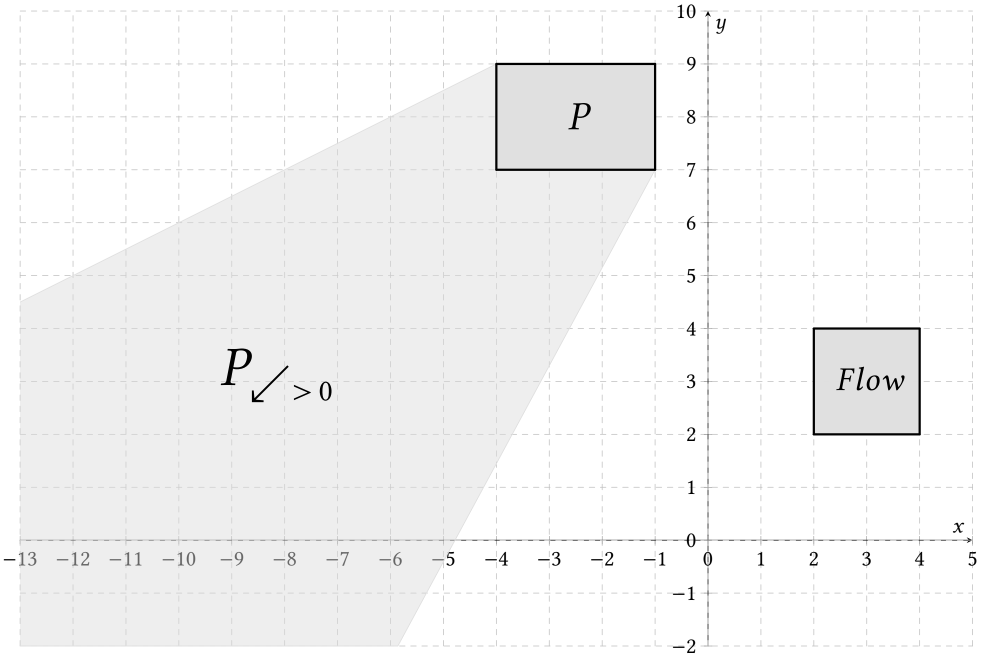


Figure 2.10: Geometric intuition of the positive pre-flow $P_{\swarrow > 0}$.

Let $A \subseteq \mathbb{R}^n$ be a polyhedron and $B \subseteq \mathbb{R}^n$ a convex polyhedron. Then:

$$\mathit{reach}^0(A, B) = A \cap \mathrm{cl}(B) \cap (B_{\swarrow > 0})$$

To compute reach^+, we use the *May Reach While Avoiding* operator, denoted $\mathit{RWA}^{\mathrm{m}}(Y, Z)$, which collects all points from which an admissible trajectory can reach the set Y while avoiding the set Z. Formally:

$$\mathit{RWA}^{\mathrm{m}}(Y, Z) \coloneqq \left\{x \in \mathbb{R}^n \mid \exists f \in \mathit{Traj}(x),\ t \geq 0.\ f(t) \in Y \wedge \forall t' \in [0, t).\ f(t') \in Y \cup \overline{Z}\right\}$$

An effective algorithm for computing $\mathit{RWA}^{\mathrm{m}}$ through symbolic polyhedral operations is presented in [9] and implemented in the tool NYCS.

Let $A \subseteq \mathbb{R}^n$ be a polyhedron and $Q \subseteq \mathbb{R}^n$ a convex polyhedron. Then:

$$\mathit{reach}^+(A, Q) = \bigcup_{P \in \mathit{Patch}(A)} \mathit{RWA}^{\mathrm{m}}(T_P, A) \quad \text{where} \quad T_P \coloneqq P \cap (\mathrm{cl}(P) \cap Q)_{\swarrow > 0}$$

2.5.4 Split Function

It is sometimes useful to decompose the result of $\mathit{reach}^\flat(A, B)$, which is in general a non-convex polyhedron, into convex components, each contained in one of the patches of A. To this aim, we define the function $\mathit{split}(A', A)$ where A' is a subset of A. The function returns a finite set of pairs $\{(P_i, X_i)\}_{i=1}^n$ such that:

- each P_i and X_i is a convex polyhedron with $X_i \subseteq P_i$,

- each P_i is a patch of A,
- $A' = \bigcup_{i=1}^{n} X_i$.

The function can be implemented using standard Boolean operations on polyhedra.

2.5.5 Computational Complexity of Reach Operators

The computation of the reachability operators $\mathit{reach}^\flat$ relies on symbolic operations over polyhedra, which are known to be superpolynomial in the worst case [23]. A rough measure of complexity can be obtained by counting the number of such symbolic operations. The operator $\mathit{reach}^0(A,B)$ involves a constant number of geometric operations: intersections, closures, and positive time-elapse [20]. The computation of $\mathit{reach}^+(A,B)$, instead, requires at most $|\mathit{Patch}(A)|$ calls to the operator $\mathit{RWA}^{\mathrm{m}}$. According to the complexity analysis in [9], computing $\mathit{RWA}(Y,Z)$ involves at most $k \cdot m^{O(m)}$ symbolic operations, where k and m are the number of convex patches of Y and Z, respectively. Moreover, the output of $\mathit{RWA}^{\mathrm{m}}$ contains at most $m^{O(m)}$ patches. Since B is assumed to be convex (i.e., one patch), we have that $\mathit{reach}^+(A,B)$ requires at most $m^{O(m)}$ operations, where m is the number of patches of A, and its result contains at most $m^{O(m)}$ patches.

—3—

Linear Temporal Logics

CONTENTS: **3.1 Syntax and Operators. 3.2 Linear Temporal Logic Semantics. 3.3 Real-Time Logic Semantics. 3.4 Non-Recurrent Fragment of** `RTL`**.**

Linear Temporal Logic (`LTL`), introduced by Pnueli [24], is a well-established formalism for expressing temporal properties of reactive systems. It allows one to formally specify requirements such as safety and liveness properties. Traditionally, the semantics of `LTL` is defined over infinite discrete traces capturing the evolution of system states in a purely discrete setting. Since our focus is on continuous dynamics, we extend this perspective by also considering a semantics based on real-valued signals, as explored in previous works [17].

Besides the classic semantics on infinite-time behaviours, for both the discrete and continuous versions, we shall also consider finite-time fragments, where formulae are interpreted over finite traces and finite-time signals, respectively. In Chapter 4, we describe how the discrete and the continuous semantics are related—a connection that we exploit to reduce the verification of continuous-time properties to a combination of verification of discrete properties and geometric reasoning.

3.1 Syntax and Operators

LTL formulae are built up in the usual way from the temporal operators and logical connectives, according to the following grammar:

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi,$$

where p is an atomic proposition in AP. The length of a formula φ is denoted by $|\varphi|$.

3.2 Linear Temporal Logic Semantics

We consider three semantic variants of Linear Temporal Logic: $\mathtt{LTL}_f$, $\mathtt{LTL}_\omega$, and $\mathtt{LTL}_\infty$. In $\mathtt{LTL}_f$, formulae are interpreted over *non-empty finite words* $\omega = \omega_0\omega_1 \ldots \omega_n$; in $\mathtt{LTL}_\omega$, over *infinite words* $\omega = \omega_0\omega_1 \ldots$; and in $\mathtt{LTL}_\infty$, over *both finite and infinite words.* To uniformly refer to these semantic variants, we use the symbol $\varkappa \in \{\mathsf{fin}, \omega, \infty\}$. The same notation is adopted for $\mathtt{RTL}_\varkappa$ in Section 3.3.

To define the semantics uniformly across these fragments, we introduce the following notation for the set of words:

$$\Sigma^\varkappa = \begin{cases} \Sigma^* & \text{if } \varkappa = f \\ \Sigma^\omega & \text{if } \varkappa = \omega \\ \Sigma^* \cup \Sigma^\omega & \text{if } \varkappa = \infty \end{cases}$$

where $\Sigma = 2^{AP}$, and the parameter $\varkappa$ selects the appropriate fragment. Given a word $\omega \in \Sigma^\varkappa$ and an index $i \geq 0$, we denote by $\omega_{\geq i}$ the *suffix* of ω starting from position i. The satisfaction relation $\omega \models_\varkappa \varphi$ is defined as follows:

- $\omega \models_\varkappa \varphi$, for $\varphi \in AP$, if and only if $\varphi \in \omega_0$;
- $\omega \models_\varkappa \neg\varphi$ if and only if $\omega \not\models_\varkappa \varphi$;
- $\omega \models_\varkappa \varphi_1 \wedge \varphi_2$ if and only if $\omega \models_\varkappa \varphi_1$ and $\omega \models_\varkappa \varphi_2$;
- $\omega \models_\varkappa \mathsf{X}\,\varphi$ if and only if $|\omega| > 1$ and $\omega_{\geq 1} \models_\varkappa \varphi$;
- $\omega \models_\varkappa \varphi_1 \,\mathsf{U}\, \varphi_2$ if and only if there exists $i \geq 0$ such that $\omega_{\geq i} \models_\varkappa \varphi_2$ and, for all j such that $0 \leq j < i$, it holds $\omega_{\geq j} \models \varphi_1$;
- $\omega \models_\varkappa \varphi_1 \,\mathsf{R}\, \varphi_2$ if and only if for all $i \geq 0$, it holds $\omega_{\geq i} \models_\varkappa \varphi_2$ or there exists $j < i$ such that $\omega_{\geq j} \models \varphi_1$.

3.3 Real-Time Logic Semantics

To define the semantics of $\mathtt{RTL}_\varkappa$ where $\varkappa \in \{f, \omega, \infty\}$, we consider three classes of time intervals over the non-negative real numbers $\mathbb{R}_{\geq 0}$: bounded intervals, unbounded intervals, and the union of both. We define the following sets of intervals:

$$\begin{aligned} \mathcal{I}^f &= \{I \subseteq \mathbb{R}_{\geq 0} \mid \sup(I) < \infty\} && \text{(bounded intervals)} \\ \mathcal{I}^\omega &= \{I \subseteq \mathbb{R}_{\geq 0} \mid \sup(I) = \infty\} && \text{(unbounded intervals)} \\ \mathcal{I}^\infty &= \mathcal{I}^f \cup \mathcal{I}^\omega && \text{(both bounded and unbounded intervals)} \end{aligned}$$

Given these classes of intervals, we define the corresponding sets of signals over atomic propositions AP. For each $\varkappa \in \{f, \omega, \infty\}$, the set of signals $Sig^\varkappa$ is defined as:

$$Sig^\varkappa = \left\{\sigma : I \to 2^{AP} \,\middle|\, I \in \mathcal{I}^\varkappa\right\}$$

Depending on the class $\varkappa$, the domain of the signal may be bounded, unbounded, or both. The semantics of $\mathtt{RTL}_{\varkappa}$ formulae is given with respect to a signal $\sigma \in Sig^{\varkappa}$. A signal is *left-open* if its domain interval I is left-open, namely if $\inf(I) \notin I$; otherwise, it is *left-closed*. The semantic conditions are defined as follows:

- $\sigma \models_{\varkappa} p$, for $p \in AP$, if and only if:
  - $p \in \sigma(\inf(I))$, if $\inf(I) \in I$;
  - there exists $d \in I$ such that $p \in \sigma(d')$, for all $d' \in I$ with $d' \leq d$, if $\inf(I) \notin I$.
- $\sigma \models_{\varkappa} \mathsf{X}\,\varphi$ if and only if there exists $d \in I \setminus \{\inf(I)\}$ such that $\sigma_{\geq d'} \models_{\varkappa} \varphi$ for all $d' \in I \setminus \{\inf(I)\}$ with $d' \leq d$.
- $\sigma \models_{\varkappa} \varphi_1 \,\mathsf{U}\, \varphi_2$ if and only if there exists $d \in I$ such that $\sigma_{\geq d} \models_{\varkappa} \varphi_2$ and $\sigma_{\geq d'} \models_{\varkappa} \varphi_1$ for all $d' \in I$ with $d' < d$.
- $\sigma \models_{\varkappa} \varphi_1 \,\mathsf{R}\, \varphi_2$ if and only if for all $d \in I$, it holds that $\sigma_{\geq d} \models_{\varkappa} \varphi_2$ or there exists $d' \in I$ with $d' < d$ such that $\sigma_{\geq d'} \models_{\varkappa} \varphi_1$.

3.4 Non-Recurrent Fragment of RTL

We shall also consider the *non-recurrent fragment* of RTL

$$, whichconsistsofformulaeinpositivenormalformwhere, ineachreleasesubformulaoftheform$$

$\varphi_1 \,\mathsf{R}\, \varphi_2$, the second argument φ_2 is required to be a *propositional formula* (i.e., a boolean combination of atomic propositions). This fragment allows for temporal operators to occur, but it disallows temporal properties to be required *continuously*, except when the requirement is purely propositional. For instance, the formula $\bot \,\mathsf{R}\,(p \wedge q)$, which is equivalent to $\mathsf{G}\,(p \wedge q)$, belongs to the fragment, since $(p \wedge q)$ is propositional. In contrast, the formula $\bot \,\mathsf{R}\,(\mathsf{F}p)$, which is equivalent to $\mathsf{GF}\,p$, does *not* belong to the fragment, because the temporal formula $\mathsf{F}p$ is required to hold at all time points. The non-recurrent fragment we consider is an extension of the *flat fragment* of LTL introduced in [25], where, in addition, the first argument of a until-formula (U) is also required to be propositional.

–4–

Solving the Model Checking Problem for $\mathtt{RTL}_\omega/\mathtt{RTL}_f$

CONTENTS: **4.1 The Model-Checking Problem.** 4.1.1 Existential Denotation – 4.1.2 Universal Denotation. **4.2 Decidability Scenarios for each Semantics.** **4.3 Discretisation of an `RTL` Formula.** 4.3.1 Discretising Signals – 4.3.2 Discretising Formulae – 4.3.3 Automaton Construction for $\mathtt{RTL}_f/\mathtt{RTL}_\omega$. **4.4 Finite-Time Semantics.** 4.4.1 General Case – 4.4.2 Omnidirectional Flow Scenario. **4.5 Infinite-Time Semantics.** 4.5.1 Omnidirectional Flow Scenario – 4.5.2 Non-Recurrent $\mathtt{RTL}_\omega$ and Compact Flow Scenario. **4.6 Maximal Semantics (may, must).** 4.6.1 The *brink* Atomic Proposition – 4.6.2 Omnidirectional Flow Scenario – 4.6.3 Forced Motion and Bounded Invariant Scenario – 4.6.4 Non-Recurrent `RTL` and Compact Flow Scenario.

4.1 The Model-Checking Problem

We are interested in solving the model-checking problem for an `RTL` formula φ against a polyhedral system and under one of the semantics $\gamma \in \{\text{fin}, \text{inf}, \text{may}, \text{must}\}$, as described in Section 2.4.2. For a point $x \in \mathit{Inv} \subseteq \mathbb{R}^n$, the problem requires to decide whether $x \models_\gamma \phi$, namely if there exists a trajectory in $\mathit{Traj}_\gamma(x)$ from that point whose induced signal satisfies the formula. Formally, the relation $\models_\gamma$ is defined as follows:

$$x \models_\gamma \phi \quad \text{if and only if} \quad \exists f \in \mathit{Traj}_\gamma(x) \text{ such that } \sigma_f \models \phi$$

4.1.1 Existential Denotation

Rather than directly addressing the model-checking problem for a single, specific initial point x as formulated in Section 4.1, we approach a more general task. Our goal is to determine the set of *all* initial points within the system's invariant *Inv* from which the `RTL` formula φ can be satisfied under the chosen semantics γ.

Given an `RTL` formula φ, a polyhedral system $\mathcal{P}$ defined over the same set of atomic propositions, and a semantics $\gamma \in \{\text{fin}, \text{inf}, \text{may}, \text{must}\}$, the *existential denotation* of φ on $\mathcal{P}$ with respect to the semantics γ, denoted by $[\![\varphi]\!]^{\exists}_{\gamma}$, is the set of points $x \in Inv$ such that $x \models_{\gamma} \varphi$. Formally:

$$[\![\varphi]\!]^{\exists}_{\gamma} = \{x \in Inv \subseteq \mathbb{R}^n \mid x \models_{\gamma} \varphi\}$$

Computing this set is a more general problem than checking satisfaction for a single point: once $[\![\varphi]\!]^{\exists}_{\gamma}$ is known, the original model-checking problem for any specific point $x \in Inv$ reduces to checking whether $x \in [\![\varphi]\!]^{\exists}_{\gamma}$.

4.1.2 Universal Denotation

Complementary to the existential denotation, we can define the *universal denotation* of a formula `RTL` φ. While the existential denotation identifies points from which *at least one* trajectory satisfies φ, the universal denotation captures the set of points from which *all* possible trajectories satisfy φ under the given semantics γ. Formally, given the same `RTL` formula φ, polyhedral system $\mathcal{P}$, and semantics $\gamma \in \{\text{fin}, \text{inf}, \text{may}, \text{must}\}$, the *universal denotation* of φ on $\mathcal{P}$ with respect to γ, denoted by $[\![\varphi]\!]^{\forall}_{\gamma}$, is the set of points $x \in Inv$ such that *all* trajectories starting from x satisfy φ:

$$[\![\varphi]\!]^{\forall}_{\gamma} = \{x \in Inv \subseteq \mathbb{R}^n \mid \forall f \in Traj_{\gamma}(x) \text{ it holds that } \sigma_f \models \varphi\}$$

Clearly, the universal denotation $[\![\varphi]\!]^{\forall}_{\gamma}$ of a formula can easily be computed by duality, that is:

$$[\![\varphi]\!]^{\forall}_{\gamma} = Inv \setminus [\![\neg\varphi]\!]^{\exists}_{\gamma}$$

4.2 Decidability Scenarios for each Semantics

Having defined the model-checking problem and the concept of existential denotation, we now turn to the fundamental question of decidability: under which conditions can we algorithmically determine the set $[\![\varphi]\!]^{\exists}_{\gamma}$? The problem of computing the existential denotation $[\![\varphi]\!]^{\exists}_{\gamma}$ for an `RTL` formula φ and a polyhedral system $\mathcal{P}$ is decidable for all semantics $\gamma \in \{\text{fin}, \text{inf}, \text{may}, \text{must}\}$, although sometimes requiring additional assumptions either on the system $\mathcal{P}$ or on the class of formulae considered. Specifically, we establish decidability in the following scenarios:

1. **Finite-Time Semantics ($\gamma = \text{fin}$):** The problem is decidable without any further restrictions on the polyhedral system $\mathcal{P}$ or the `RTL`$_\text{f}$ formula φ. This general decidability result for the finite-time case will be explored in Section 4.4.

2. **Infinite-Time Semantics (γ = inf):** Decidability is guaranteed under specific assumptions:

    (a) *Omnidirectional Flow:* This scenario assumes the origin is contained within the interior (see Section 2.1.2) of the *Flow* constraint associated with $\mathcal{P}$ (i.e., $0 \in int(Flow)$). Intuitively, this allows the system state to remain stationary or move in any direction. No restrictions are placed on the $\mathtt{RTL}_\omega$ formula φ. This case is addressed in Section 4.5.1.

    (b) *Non-Recurrent $\mathtt{RTL}_\omega$ and Compact Flow:* Here, decidability holds if we restrict the formulae to the non-recurrent fragment of $\mathtt{RTL}_\omega$ and assume the *Flow* of $\mathcal{P}$ is a compact convex polyhedron. This scenario is discussed in Section 4.5.2.

3. **Maximal Semantics ($\gamma \in$ {may, must}):** For the maximal semantics, described in Section 4.6, decidability is shown under the following conditions:

    (a) *Omnidirectional Flow:* Similar to the infinite-time case, if $0 \in int(Flow)$, the problem is decidable without restrictions on the $\mathtt{RTL}_\infty$ formula φ. This scenario is discussed in Section 4.6.2.

    (b) *Forced Motion and Bounded Invariant:* Decidability holds if the system exhibits forced motion, meaning the origin is not in the closure of the flow ($0 \notin cl(Flow)$), and the invariant *Inv* is bounded. Under forced motion, the system cannot stop or move arbitrarily slowly while inside the invariant. The boundedness of *Inv* ensures that any trajectory must eventually reach the boundary of the invariant and stop, effectively reducing the analysis only to finite-time trajectories. No restrictions are placed on the $\mathtt{RTL}_\infty$ formula φ. This scenario is discussed in Section 4.6.3).

    (c) *Non-Recurrent $\mathtt{RTL}_\infty$ and Compact Flow:* Analogous to the infinite-time case, decidability is achieved if only non-recurrent $\mathtt{RTL}_\infty$ formulae are considered and the flow *Flow* is a closed polyhedron. This scenario is discussed in Section 4.6.4.

It is important to note that the model-checking problem for the semantics $\gamma \in$ {inf, may, must} remains an open problem in the general case, i.e. without imposing restrictions on the polyhedral system $\mathcal{P}$ or the $\mathtt{RTL}_\omega/\mathtt{RTL}_\infty$ formula φ.

4.3 Discretisation of an RTL Formula

To address the model-checking problem of a polyhedral system $\mathcal{P}$ against an RTL specification φ, our core strategy is to reduce this continuous-time problem to an equivalent decision problem within a discrete-time framework. Specifically, we leverage the well-established theories of classic ω-regular Linear Temporal Logic ($\mathtt{LTL}_\omega$) and its finite-trace variant $\mathtt{LTL}_f$.

The core of this reduction is the construction of a mapping, denoted $\mathtt{dsc}(\cdot)$, that transforms an $\mathtt{RTL}_\omega/\mathtt{RTL}_\mathrm{f}$ formula φ over a set of atomic propositions AP into a corresponding $\mathtt{LTL}_\omega$ (or $\mathtt{LTL}_f$) formula, $\mathtt{dsc}(\varphi)$. This target discrete formula operates on an extended set of atomic propositions, such that a signal σ satisfies φ *iff* the discrete traces induced by σ satisfy $\mathtt{dsc}(\varphi)$. In order to do this, we first need to define and characterise the discrete versions of signals (Section 4.3.1) and formulae (Section 4.3.2).

4.3.1 Discretising Signals

The first step in bridging the continuous and discrete domains is to define how a continuous signal σ can be represented as a discrete sequence, or *trace*. Recall from Section 2.4.4 that a *time slicing* $\tau = \{t_i\}_{i=0} \in TS(\sigma)$ for a signal $\sigma : I \longrightarrow 2^{AP}$ decomposes its time domain I into a sequence of slices. This sequence corresponds to an alternation of singular time points t_i and open time intervals (t_i, t_{i+1}). The sequence of slices is finite if σ is a finite-time signal and infinite otherwise.

To formally map continuous time to discrete steps based on the slicing τ, we introduce the function $\mathsf{slice}^\tau_\sigma : I \longrightarrow \mathbb{N}$. This function unambiguously associates each time instant $t \in I$ with the index of the slice it belongs to within the decomposition induced by τ. Formally:

$$\mathsf{slice}^\tau_\sigma(t) = \begin{cases} 2i & \text{if } t = t_i \text{ for some } i \geq 0 \\ 2i+1 & \text{if } t \in (t_i, t_{i+1}) \text{ for some } i \geq 0 \end{cases}$$

Note that even indices correspond to the singular time points t_i, while odd indices correspond to the open intervals (t_i, t_{i+1}).

Using this indexing, we define the *discrete trace* $trc(\sigma, \tau)$ associated with the signal σ and the slicing τ. This trace essentially “lumps together” the behaviour within each open interval (t_i, t_{i+1}) into a single discrete step and intersperses these steps with the information from the singular points t_i that separate them.

To maintain the crucial distinction between elements derived from open intervals and those derived from singular points within the discrete trace, we introduce an auxiliary atomic proposition, denoted $sing$. This proposition holds true ($sing$ is included in the set of propositions) at a discrete step if and only if that step corresponds to a singular time point t_i from the slicing τ.

Let $\alpha_i \subseteq AP$ denote the set of atomic propositions that hold constantly throughout the open interval (t_i, t_{i+1}). The discretisation $trc(\sigma, \tau)$ is then constructed as a word (sequence) over the alphabet $2^{AP \cup \{sing\}}$. This word is finite if σ is a finite-time signal and infinite otherwise. We use $trc(\sigma, \tau)_j$ to denote the j-th symbol (a set of propositions) of this discrete trace.

The precise definition depends slightly on whether the signal's domain I is left-closed or left-open.

- **For a left-closed signal** $\sigma : I \longrightarrow 2^{AP}$ (where $I = [t_0, \dots)$ or $I = [t_0, t_N]$) and a discrete index $j \in \mathrm{rng}(\mathrm{slice}^{\tau}_{\sigma})$:

$$trc(\sigma, \tau)_j \triangleq \begin{cases} \sigma(t_i) \cup \{sing\}, & \text{if } j \text{ is even and } i = j/2 \\ \alpha_i & \text{if } j \text{ is odd and } i = (j-1)/2 \end{cases}$$

- **For a left-open signal** $\sigma : I \longrightarrow 2^{AP}$ (where $I = (t_0, \dots)$ or $I = (t_0, t_N]$) and a discrete index $j \in \mathrm{rng}(\mathrm{slice}^{\tau}_{\sigma})$:

$$trc(\sigma, \tau)_j \triangleq \begin{cases} \alpha_i & \text{if } j \text{ is even and } i = j/2 \\ \sigma(t_i) \cup \{sing\} & \text{if } j \text{ is odd and } i = (j+1)/2 \quad \text{(assuming } t_i \in I) \end{cases}$$

Furthermore, it is immediate from the construction that any trace $\mathrm{trc}(\sigma, \tau)$ generated from a signal σ possesses specific structural properties related to the auxiliary proposition $sing$:

(a) singular and open intervals alternate throughout the trace;

(b) the trace starts in a singular interval *iff* the underlying signal is left-closed;

(c) the trace ends in a singular interval *iff* the underlying signal is right-closed.

These structural properties can be formally captured using `LTL` operators as follows:

Proposition 1 ([15]). *For any signal* $\sigma : I \longrightarrow 2^{AP}$ *and any time slicing* $\tau \in TS(\sigma)$, *the discrete trace* $\mathrm{trc}(\sigma, \tau)$ *satisfies:*

$$\mathrm{trc}(\sigma, \tau) \models \mathrm{G}((sing \leftrightarrow \mathrm{X}\neg sing) \vee last)$$

where $last \triangleq \neg \mathrm{X}\top$ *characterises the final state of a finite trace. Furthermore:*

- σ *is left-closed if and only if* $\mathrm{trc}(\sigma, \tau) \models sing$.
- *If* σ *is finite,* σ *is right-closed if and only if* $\mathrm{trc}(\sigma, \tau) \models F(last \wedge sing)$.

Finally, given a trajectory f, we extend this definition naturally: the discrete trace associated with the trajectory f and slicing τ is defined as the trace of its induced signal σ_f:

$$trc(f, \tau) \triangleq trc(\sigma_f, \tau)$$

4.3.2 Discretising Formulae

Having established how to obtain a discrete trace $\mathsf{trc}(\sigma, \tau)$ from a continuous signal σ and a slicing τ, we now define the corresponding transformation for $\mathtt{RTL}_\omega$ formulae. We introduce a mapping $\mathtt{dsc}(\cdot)$ that translates an $\mathtt{RTL}_\omega$ formula φ over the atomic propositions AP into an $\mathtt{LTL}_\omega$ formula $\mathtt{dsc}(\varphi)$ over the extended set $AP \cup \{sing\}$.

This translation is designed such that the satisfaction of φ by a continuous signal σ corresponds directly to the satisfaction of $\mathtt{dsc}(\varphi)$ by the discrete traces induced by σ. The mapping exploits the information provided by the *sing* proposition within the trace to correctly interpret the temporal structure according to the original `RTL` semantics.

Formally, the transformation $\mathtt{dsc}(\cdot)$ is defined recursively as follows:

$$\begin{aligned}
\mathtt{dsc}(p) &\triangleq p \\
\mathtt{dsc}(\neg\varphi) &\triangleq \neg\mathtt{dsc}(\varphi) \\
\mathtt{dsc}(\varphi_1 \wedge \varphi_2) &\triangleq \mathtt{dsc}(\varphi_1) \wedge \mathtt{dsc}(\varphi_2) \\
\mathtt{dsc}(\varphi_1 \vee \varphi_2) &\triangleq \mathtt{dsc}(\varphi_1) \vee \mathtt{dsc}(\varphi_2)
\end{aligned}$$

$$\begin{aligned}
\mathtt{dsc}(\mathsf{X}\,\varphi) &\triangleq (sing \wedge \mathsf{X}\,\mathtt{dsc}(\varphi)) \vee (\neg sing \wedge \mathtt{dsc}(\varphi)) \\
\mathtt{dsc}(\varphi_1 \,\mathsf{U}\, \varphi_2) &\triangleq \mathtt{dsc}(\varphi_1) \,\mathsf{U}\, ((sing \vee \mathtt{dsc}(\varphi_1)) \wedge \mathtt{dsc}(\varphi_2)) \\
\mathtt{dsc}(\varphi_1 \,\mathsf{R}\, \varphi_2) &\triangleq \mathtt{dsc}(\varphi_1) \,\mathsf{R}\, ((\neg sing \wedge \mathtt{dsc}(\varphi_1)) \vee \mathtt{dsc}(\varphi_2))
\end{aligned}$$

The fundamental result connecting the continuous semantics with the discrete semantics of the translated formula is captured by the following theorem.

Theorem 1 ([15]). *For all RTL_ω formulae φ, signals σ, and time slicings $\tau \in TS(\sigma)$, it holds that:*

$$\sigma \models \varphi \quad \textit{if and only if} \quad \mathrm{trc}(\sigma, \tau) \models \mathrm{dsc}(\varphi)$$

The correctness of the transformation $\mathtt{dsc}(\cdot)$, as stated in Theorem 1, relies on detailed proofs demonstrating the equivalence for all operators. These formal proofs can be found in [15]. This theorem is the cornerstone of our approach, as it allows us to reduce the $\mathtt{RTL}_\omega$ model-checking problem to an equivalent $\mathtt{LTL}_\omega$ model-checking problem on discrete traces.

4.3.3 Automaton Construction for $\mathtt{RTL}_\mathsf{f}/\mathtt{RTL}_\omega$

Leveraging the discretisation mapping $\mathtt{dsc}(\cdot)$ described in Section 4.3.2 and the equivalence established in Theorem 1, our goal is now to construct automata that recognise exactly the discrete traces $\mathsf{trc}(\sigma, \tau)$ corresponding to signals σ satisfying a given $\mathtt{RTL}_\omega/\mathtt{RTL}_\mathrm{f}$ formula φ.

As described in Proposition 1, valid traces generated by $\mathsf{trc}(\sigma, \tau)$ exhibit a strict alternation between states representing singular intervals (where *sing* holds) and states representing open intervals (where $sing$ does not hold). The raw translated formula $\mathtt{dsc}(\varphi)$ does not inherently enforce this structure. Therefore, we incorporate this structural requirement directly into the `LTL` formulae for which we build automata.

Let φ be the original $\mathtt{RTL_f}/\mathtt{RTL}_\omega$ formula. We define the following $\mathtt{LTL}_\omega/\mathtt{LTL_f}$ formulae over the alphabet $\Sigma = 2^{AP \cup \{sing\}}$:

$$\widehat{\varphi} \triangleq \mathtt{dsc}(\varphi) \wedge \mathsf{G}(sing \leftrightarrow \mathsf{X}\neg sing) \tag{4.1}$$

$$\widehat{\varphi}^{\mathrm{fin}} \triangleq \widehat{\varphi} \wedge \mathsf{F}(last \wedge sing) \tag{4.2}$$

Here, $\widehat{\varphi}$ combines the original property $\mathtt{dsc}(\varphi)$ with a constraint $\mathsf{G}(sing \leftrightarrow \mathsf{X}\neg sing)$ that enforces the required alternation between singular and non-singular observables. This formula is targeted towards infinite traces ($\mathtt{LTL}_\omega$). The formula $\widehat{\varphi}^{\mathrm{fin}}$, designed for finite traces ($\mathtt{LTL_f}$), includes the constraints of $\widehat{\varphi}$ and adds the condition $\mathsf{F}(last \wedge sing)$. This ensures the trace is finite (reaches a state satisfying $last$) and ends in a singular state, characteristic of traces from right-closed signals.

The existence of automata corresponding to these $\mathtt{LTL}_\omega/\mathtt{LTL}_f$ formulae is guaranteed by standard results in automata theory:

Theorem 2 ([18, 19])**.** *For every LTL_ω (resp., LTL_f) formula ψ over an alphabet Σ, there exists a Büchi automaton (resp., a Nondeterministic Finite Automaton) $\mathcal{A}_\psi$ over Σ that accepts exactly the set of infinite (resp., finite) words satisfying ψ.*

Applying Theorem 2 to our specific formulae $\widehat{\varphi}$ and $\widehat{\varphi}^{\mathrm{fin}}$ allows us to construct the required automata. We denote by A_ϕ (resp., A_ϕ^{fin}) the Büchi automaton (resp., finite automaton) for $\widehat{\varphi}$ (resp., $\widehat{\varphi}^{\mathrm{fin}}$) provided by Theorem 2, with components $(2^{AP}, S, \delta, \lambda, S_0, S_F)$. All components have their standard meaning. In particular, λ labels each state $s \in S$ with a subset of $AP \cup \{sing\}$. For convenience, we may write $[\![s]\!]$ for $[\![\lambda(s) \cap AP]\!]$ to denote the polyhedron interpreting the set of propositions labelling s.

As an immediate corollary of the above Theorem 2 and Proposition 1, we have the following theorem.

Theorem 3 ([15])**.** *For all RTL_ω/RTL_f formulae ϕ, the Büchi automaton A_ϕ (resp., finite automaton A_ϕ^{fin}) satisfies the following property: for all infinite-time (resp., finite-time) signals σ, it holds that $\sigma \models \phi$ iff $trc(\sigma, \tau)$ is recognised by A_ϕ (resp., A_ϕ^{fin}), for all time slicings $\tau \in TS(\sigma)$.*

4.4 Finite-Time Semantics

In the following, for each scenario, we define a finite abstraction of the model, called a *polyhedral abstraction*, i.e. a finite graph whose nodes are labelled with a (possibly non-convex) polyhedron which is a subset of some combination of atomic propositions (*a.k.a.* an observable). Formally, a polyhedral abstraction is a tuple $G = (V, E, \mathrm{pts}, \mathrm{obs})$, where (V, E) is a finite graph, and, for each node $v \in V$, the set $\mathrm{pts}(v)$ is a non-empty polyhedron in $\mathbb{R}^n$, $\mathrm{obs}(v) \subseteq AP$, and it holds $\mathrm{pts}(v) \subseteq [\![\mathrm{obs}(v) \cap AP]\!]$. Polyhedral abstractions are meant to represent the relevant behaviours of polyhedral systems.

4.4.1 General Case

To define the first polyhedral abstraction, we introduce some auxiliary notation. We use the symbol $\flat \in \{0, +\}$ to distinguish singular from open time intervals, and write $\neg\flat$ for its complement. States in the abstraction correspond to finite sequences of observable patches, alternating 0's and $+$'s.

Assuming each observable $\alpha \subseteq AP$ is represented as a finite set of convex polyhedra $[\![\alpha]\!]$, we define the set of patches as:

$$\mathcal{P} \coloneqq \bigcup_{\alpha \subseteq AP} [\![\alpha]\!].$$

For any patch $P \in \mathcal{P}$, we denote by α_P the observable it belongs to.

Given a finite sequence $\Delta \in \mathcal{P}^*$, we define the functions $\mathit{Trav}^\flat(\Delta)$, collecting the set of initial points of trajectories that traverse Δ with interval type $\flat$. Denoting by ε the empty sequence, we define $\mathit{Trav}^\flat$ recursively as:

$$\begin{aligned} \mathit{Trav}^\flat(\varepsilon) &= \mathbb{R}^n, \\ \mathit{Trav}^0(P\Delta) &= \boldsymbol{reach}^0(P, \mathit{Trav}^+(\Delta)), \\ \mathit{Trav}^+(P\Delta) &= P \cap \boldsymbol{reach}^+([\![\alpha_P]\!], \mathit{Trav}^0(\Delta)). \end{aligned}$$

Since $\boldsymbol{reach}^0$ and $\boldsymbol{reach}^+$ produce polyhedra, $\mathit{Trav}^\flat(\Delta)$ is always a polyhedron.

Example. As an example, consider:

$$\mathit{Trav}^0(PQR) = \boldsymbol{reach}^0\left(P, Q \cap \boldsymbol{reach}^+\left([\![\alpha_Q]\!], \boldsymbol{reach}^0(R, \mathbb{R}^n)\right)\right)$$

which includes points in P from which a trajectory reaches Q immediately and then R, spending some time in the observable of Q.

Polyhedral abstraction. The abstraction $G = (V, E, \mathrm{pts}, \mathrm{obs})$ is defined as follows:

- **Nodes:** V contains one node for each pair $(\flat, \Delta)$, where Δ is a non-empty patch sequence of length at most H_φ and $\mathit{pts}((\flat, \Delta)) \neq \emptyset$.
- **Edges:** For every node $(\flat, P\Delta)$, there is an edge to $(\neg\flat, \Delta)$.
- **Observables:** $\mathit{obs}((0, P\Delta)) = \alpha_P \cup \{\mathit{sing}\}$, and $\mathit{obs}((+, P\Delta)) = \alpha_P$.
- **Points:** $\mathit{pts}((\flat, \Delta)) = \mathit{Trav}^\flat(\Delta)$.

Note that the denotations $\mathit{pts}((\flat, \Delta))$ may overlap. The resulting graph (V, E) is a forest, with roots corresponding to sequences of length 1.

Synchronous Product between G and $\mathcal{A}_\varphi^{\mathsf{fin}}$

To solve the existential model-checking problem under finite-time semantics, we compute the synchronous product of the finite automaton $\mathcal{A}_\varphi^{\mathrm{fin}}$ representing the formula φ and the polyhedral abstraction G. Each state of the product automaton is a pair (s, v), where s is a state of $\mathcal{A}_\varphi^{\mathrm{fin}}$ and $v \in V$ is a node of G. Transitions in the product automaton are defined by synchronizing the labels of $\mathcal{A}_\varphi^{\mathrm{fin}}$ with the observables $\mathrm{obs}(v)$ from G. Initial and accepting states are determined based on the components from $\mathcal{A}_\varphi^{\mathrm{fin}}$. The denotation of each product state (s, v) is given by $\mathrm{pts}(v)$. By analyzing the product automaton, we can determine the set of initial points $x \in \mathbb{R}^n$ from which there exists a trajectory satisfying the formula φ under finite-time semantics. This is achieved by identifying all initial states of the product automaton that can reach an accepting state and taking the union of their denotations.

An On-The-Fly Algorithm for the Finite-Time Semantics

We describe the on-the-fly algorithm that solves the existential denotation problem for $\mathtt{RTL}_\mathrm{f}$ without computing the polyhedral abstraction in advance. This is significantly more efficient than explicitly constructing the full polyhedral abstraction in the general case, as the latter quickly becomes intractable even for very small inputs. The basic idea is that it explores backward each path of the automaton for the discretised formula $\widehat{\varphi}^{\mathrm{fin}}$ (see Section 4.3.3), starting from an accepting state and, in doing so, incrementally computes the denotation of the corresponding node in the polyhedral abstraction for that path. Unless otherwise specified, we assume a fixed $\mathtt{RTL}_\mathrm{f}$ formula φ over the set of atomic propositions AP, and a fixed polyhedral system $\mathcal{P}$ on AP.

Let $\mathcal{A}_{\widehat{\varphi}}^{\mathrm{fin}}$, introduced in Section 4.3.3, the automaton that encodes the finite-time semantics of the discretised formula $\widehat{\varphi}^{\mathrm{fin}}$. The following proposition highlights some structural properties of this automaton that are instrumental to the design and correctness of the on-the-fly algorithm.

Proposition 2. *The finite automaton* $\mathcal{A}_{\widehat{\varphi}}^{fin}$ *satisfies the following properties:*

(a) the underlying graph is bipartite in (S_{sing}, S_{open})*, where* S_{sing} *is the set of all states labelled with* sing*, while* S_{open} *is its complement;*

(b) all the final states are labelled with the proposition sing*;*

(c) the initial states have no predecessors.

The main steps in the procedure to solve the existential denotation problem are illustrated by pseudo-code of the function $\exists\mathsf{Denot}(\cdot)$ in Algorithm 1. Theorem 4 at the end of this section describes the top-level invocations that start the process, which begins from a final state of the automaton and then works recursively backward towards the initial states.

To ensure termination, the algorithm keeps track of the patches associated with open states in S_{open} that have been visited in the current sequence of recursive calls. Those are the patches in which the induced trajectory must spend some positive amount of time. This information is kept in the map V , that associates with each state s the set of patches of $[\![s]\!]$ already encountered by the algorithm. The algorith also use the split function defined in Section 2.5.4.

Line **1** takes care of the base case of the algorithm, when s is an initial state. In this case, if s belongs to S_0, then it returns X itself (observe that $X = \mathit{reach}^0(X, \mathbb{R}^n)$). If, however, $s \in S_{open}$, then the correct answer contains the points in X from where a trajectory can spend some time in X. These points are precisely those contained in $\mathit{reach}^+(X, \mathbb{R}^n)$. If s is not initial, an updated map V' is computed, where the patch P is added to $V(s)$ if s is an open state (Line˜5). The for loop at Lines **6–13** iterates over the incoming edges of s. For each such edge (s', s), Line **7** sets A to the region of $[\![s']\!]$ that has *not* been already visited. Line **8** computes the set of points of A that can reach some point in X, either leaving A immediately, if s' is a singular state ($\mathit{type}(s) = 0$), or lingering in A for some time, if it is open ($\mathit{type}(s) = +$). Line **9** splits the resulting set A' into a set of distinct pairs (Q_i, Y_i), where Y_i is the maximal convex polyhedron contained in A' and in the patch Q_i of A. Each such pair (Q_i, Y_i), then, gives rise to a recursive call on the state s' with targets Y_i and Q_i at Line **11**. The results of all such calls are gathered in *Result*, which is returned at Line **14**. We conclude this section with the following correctness theorem:

Theorem 4 ([15])**.** *For all RTL_f formulae φ and polyhedral systems $\mathcal{P}$ on the same set of atomic propositions, let $\widehat{\varphi}^{fin}$ be the corresponding LTL_f formula, $\mathcal{A}_\varphi^{fin}$ be the finite automaton for $\widehat{\varphi}_{fin}$, and let:*

$$X = \bigcup_{s \in S_F} \bigcup_{P \in Patch([\![s]\!])} \exists \mathsf{Denot}(s, P, P, \emptyset).$$

Then, X is the set of points from which there exists a trajectory that satisfies φ.

The full proofs and additional details can be found in [15].

NFA Optimization to Improve On-The-Fly Algorithm Performance

In this section, we discuss a simple technique for optimising the finite automaton (NFA) to improve the performance of the on-the-fly algorithm. Given the automaton $\mathcal{A}_\varphi^{\text{fin}}$ for a formula φ, we consider the resulting automaton $\mathcal{A'}_\varphi^{\text{fin}}$ obtained by removing all outgoing edges from each final state of $\mathcal{A}_\varphi^{\text{fin}}$. The key observation is that this modification does not affect the existential denotation result. Therefore, $\mathcal{A'}_\varphi^{\text{fin}}$ produces the same existential denotation as $\mathcal{A}_\varphi^{\text{fin}}$, while having a potentially smaller structure that can lead to performance improvements in the algorithm.

If an accepting path in $\mathcal{A}_\varphi^{\text{fin}}$ is not present in $\mathcal{A'}_\varphi^{\text{fin}}$, then there must exist a prefix of that path in $\mathcal{A'}_\varphi^{\text{fin}}$ that contains its denotation. This means that the existential

Algorithm 1 Function $\exists\mathsf{Denot}(s, P, X, V)$. For simplicity, we omit from the notation two implicit arguments: the finite automaton $\mathscr{A}_{\varphi}^{\mathrm{fin}\mathit{fin}} = (2^{\widehat{AP}}, S, \delta, \lambda, S_0, S_F)$ and the polyhedral system $\mathcal{P}$.

```
input: $s \in S$;
        $P$: convex polyhedron in $\mathit{Patch}(\llbracket s \rrbracket)$;
        $X$: convex polyhedron included in $P$;
        $V$: map from states $u \in S$ to a subset of the patches of $\llbracket u \rrbracket$;
output: A polyhedron in $\mathbb{R}^n$

if $s \in S_0$ then
    return $\mathit{reach}^{\mathit{type}(s)}(X, \mathbb{R}^n)$
end if
$\mathit{Result} \leftarrow \emptyset$
$V' \leftarrow$ if $s \in S_{sing}$ then $V$ else $V[s \mapsto V(s) \cup \{P\}]$
for all state $s' \in S$ such that $(s', s) \in \delta$ do
    $A \leftarrow \llbracket s' \rrbracket \setminus V(s')$
    $A' \leftarrow \mathit{reach}^{\mathit{type}(s')}(A, X)$
    $\{(Q_1, Y_1), \ldots, (Q_n, Y_n)\} \leftarrow \mathit{split}(A', A)$
    for $i = 1, \ldots, n$ do
        $\mathit{Result} \leftarrow \mathit{Result} \cup \exists\mathsf{Denot}(s', Q_i, Y_i, V')$
    end for
end for
return $\mathit{Result}$
```

denotation problem can still be solved using the modified automaton $\mathcal{A}'^{\text{fin}}_{\varphi}$, as it captures all relevant information from $\mathcal{A}^{\text{fin}}_{\varphi}$, albeit with fewer states and transitions.

- Every accepting path in $\mathcal{A}'^{\text{fin}}_{\varphi}$ is also an accepting path in $\mathcal{A}^{\text{fin}}_{\varphi}$.
- For every accepting path in $\mathcal{A}^{\text{fin}}_{\varphi}$, there exists a corresponding non-strict prefix accepting path in $\mathcal{A}'^{\text{fin}}_{\varphi}$.

The following theorem formalises the equivalence between the original automaton $\mathcal{A}^{\text{fin}}_{\varphi}$ and the optimised automaton $\mathcal{A}'^{\text{fin}}_{\varphi}$ in terms of existential denotation:

Theorem 5. *Let* $\mathcal{A}'^{fin}_{\varphi} = (S, \delta', \lambda, S_0, S_F)$ *be the automaton obtained from* $\mathcal{A}^{fin}_{\varphi} = (S, \delta, \lambda, S_0, S_F)$ *by removing all outgoing edges from each of its final states* $s \in S_F$*. Then, the following equality holds:*

$$\bigcup_{s \in S_F} \bigcup_{P \in Patch(\llbracket s \rrbracket)} \exists \mathsf{Denot}(\mathcal{A}^{fin}_{\varphi}, s, P, P, \emptyset) = \bigcup_{s \in S_F} \bigcup_{P \in Patch(\llbracket s \rrbracket)} \exists \mathsf{Denot}(\mathcal{A}'^{fin}_{\varphi}, s, P, P, \emptyset)$$

4.4.2 Omnidirectional Flow Scenario

When the origin 0 lies in the interior of Flow, the problem becomes and simpler and a much more succinct abstraction based on geometric adjacency is sufficient. We define a *tile* as a maximal connected set of patches of the same observable within the invariant. Each tile is thus a polyhedral region. Although tiles may not be convex, any two points within the same tile are connected by an admissible trajectory that remains inside the tile.

Lemma 1 ([15])**.** *Assume* $0 \in \mathit{int}(\mathit{Flow})$*. Let* P *be a tile, and let* $x, y \in P$*. Then there exists an admissible trajectory* f *and a delay* $\delta \geq 0$ *such that* $f(0) = x$*,* $f(\delta) = y$*, and* $f(t) \in P$ *for all* $t \in (0, \delta)$*.*

The abstraction is defined as a tuple $G = (V, E, \mathit{pts}, \mathit{obs})$ with:

- **Nodes:** One node (P) for each tile P, and one node (P, Q, R) for each triple of tiles whose denotation (defined below) is non-empty.
- **Edges:** If (P) and (P, Q, R) are in V, then $((P), (P, Q, R)) \in E$. Similarly, if (R) and (P, Q, R) are in V, then $((P, Q, R), (R)) \in E$.
- **Observables:** For any node $v \in V$, if $v = (P)$ then $\mathit{obs}(v)$ is the set of atomic propositions true in P. If $v = (P, Q, R)$, then $\mathit{obs}(v)$ is the set of propositions true in Q plus the special proposition sing.
- **Points:** The denotation $\mathit{pts}((P))$ of the node (P) is the set of points in tile P. For $\mathit{pts}((P, Q, R))$, it is the set of points $x \in Q$ such that there exist $y \in P$, $z \in R$, admissible trajectories f_1, f_2 and times $t_1, t_2 \geq 0$ satisfying $f_1(0) = f_2(0) = x, f_1(t_1) = y, f_2(t_2) = z, f_1(t) \in P$, for all $t \in (0, t_1)$, and $f_2(t) \in R$, for all $t \in (0, t_2)$. Since $0 \in \mathit{int}(\mathit{Flow})$, straight-line segments are admissible. Hence, this condition is equivalent to $x \in Q \cap \mathit{cl}(P) \cap \mathit{cl}(R)$.

Only nodes with non-empty denotation are included in V. A path in G, alternating between nodes of the form (P) and (P, Q, R), abstracts a pair (f, τ), where f is a trajectory and τ is a time slicing. Nodes (P) represent time intervals where f stays within an observable; nodes (P, Q, R) represent instantaneous transitions (singular intervals). Nodes like (P, P, P) model stuttering, i.e. lingering in the same observable across multiple time slices.

4.5 Infinite-Time Semantics

Unlike in *finite-time semantics*, in the case of *infinite-time semantics*, the automaton constructed from the LTL formula is not a finite automaton, but rather a Büchi automaton. This change is necessary because infinite-time semantics considers infinite executions of the system, and only Büchi automata can recognize ω-regular languages, i.e. languages over infinite words.

4.5.1 Omnidirectional Flow Scenario

The approach for solving the omnidirectional flow scenario under infinite-time semantics closely mirrors that of the finite-time case describe in Section 4.4.2. As before, we construct the synchronous product between the omnidirectional polyhedral abstraction and the Büchi automata of the formula. However, due to the nature of the Büchi acceptance condition, the reachability analysis on the product graph changes.

Specifically, instead of identifying initial states that can reach a finite accepting state (as in the finite-time case), we now look for initial states that can reach a non-trivial connected component in the product that contains at least one accepting state of the Büchi automaton. This condition ensures the existence of an infinite path in the product where the acceptance condition is satisfied infinitely often. The denotation of the system with respect to the formula is then given by the set of initial states that can reach such an accepting non-trivial connected components.

4.5.2 Non-Recurrent $\texttt{RTL}_\omega$ and Compact Flow Scenario

The peculiarity of reasoning about the non-recurrent fragment of `RTL` under infinite-time semantics lies in the fact that such reasoning can be decomposed into two parts a generalised reachability property, which must hold on a finite prefix of the trajectory, and a safety property, which must persist throughout the infinite suffix. This decomposition allows us to address the first part using the techniques from the previous section, while the second is ensured by verifying that the trajectory eventually remains forever within a region whose observables satisfy the safety condition. Fortunately, the latter property can be verified locally as a property of the patches of the polyhedra associated with the observables.

To formalise this idea, we introduce an auxiliary proposition, stay, which holds in the patches of the observables where at least one trajectory remains indefinitely. That is, the patch is trajectory unbounded, or, using the terminology of [9], it is not $t - \mathit{bounded}$ with respect to Flow.

Formally, a convex polyhedron P is said to be $t - \mathit{bounded}$ with respect to Flow if, for every point $x \in P$ and every admissible trajectory f starting from x, there exists a time $t \geq 0$ such that $f(t) \notin P$. By [**Corollary 5.6**, 9], a convex polyhedron is t-bounded with respect to a closed convex Flow *iff* it is bounded with respect to Flow, which means that there is an admissible straight trajectory from each point of P that exits from P at some point. This last property can be efficiently tested via simple polyhedral operations as follows [**Theorem 5.8**, 9]:

$$O_P \cap \mathit{Flow} = \emptyset,$$

where O_P denotes the *characteristic cone* of the polyhedron P, i.e. the cone generated by the origin $\mathbf{0}$ and all rays of P. Hence, P is not t-bounded if there exists a direction $\hat{v} \in \mathit{Flow}$ such that $x + t \cdot \hat{v} \in P$ for all $t \geq 0$, meaning that from every point in P we can remain in P forever. Since P is convex, such a direction corresponds to a ray of P, and thus this condition holds for all $y \in P$ as well. We can now define the denotation of the stay proposition as follows:

$$[\![\mathit{stay}]\!] = \{P \in \mathsf{Patch}([\![\alpha]\!]) \mid O_P \cap \mathit{Flow} \neq \emptyset\}, \quad \alpha \subseteq \mathsf{AP}.$$

Note that if $\mathbf{0} \in \mathit{Flow}$, then the denotation of stay coincides with that of $\top$. Let $\mathcal{P}_{\mathit{stay}}$ be the polyhedral system obtained from $\mathcal{P}$ by adding the auxiliary proposition stay with the denotation defined above. We can now state the following result:

Theorem 6 ([**??**]). *For all non-recurrent* $\mathtt{RTL}_\omega$ *formulae* φ*, the following equivalence holds:*

$$\mathcal{P}, x \models^{\mathit{inf}} \varphi \quad \mathit{iff} \quad \mathcal{P}_{\mathit{stay}}, x \models^{\mathit{fin}} \varphi \wedge \mathsf{F}(\mathit{stay} \wedge \mathit{last}).$$

The formal proof of this theorem can be found in [**??**]. At this point, leveraging the result above, the model-checking problem for a non-recurrent $\mathtt{RTL}_\omega$ formula φ over the infinite-time trajectories of $\mathcal{P}$ reduces to the corresponding problem for the formula $\mathsf{F}(\mathit{stay} \wedge \mathit{last})$ over the finite-time trajectories of the polyhedral system $\mathcal{P}_{\mathit{stay}}$. This reduction allows us to reuse the techniques developed in Section 4.4.

4.6 Maximal Semantics (may, must)

Both the trajectory termination criteria “may” and “must” share the property that, in principle, they admit both finite-time and infinite-time trajectories. The model-checking problem for a formula in $\mathtt{RTL}_\infty$ when $\gamma \in \{\mathit{may}, \mathit{must}\}$ can be decomposed into the two subproblems of checking the input formula φ under finite-time semantics γ and, separately, checking it under the infinite-time one. The set of initial points satisfying the original formula is the union of the sets of points satisfying the two subproblems. This decomposition can easily be verified sound and complete thanks to the following trivial equivalence, which holds for any formula φ:

$$\varphi \equiv (\varphi \wedge \mathsf{F}\,\mathit{last}) \vee (\varphi \wedge \mathsf{G}\,\neg\mathit{last}). \tag{4.3}$$

The disjunct $\varphi \wedge \mathsf{F}\,\mathit{last}$ asks for a finite-time trajectory that satisfies φ, while $\varphi \wedge \mathsf{G}\,\neg\mathit{last}$ seeks an infinite-time trajectory satisfying the formula. The finite-time model-checking subproblem for the semantics γ can, in turn, be reduced to the finite-time model checking problem discussed in Section 4.4. Note, however, that in the cases we are considering here not all finite-time trajectories are admissible. Intuitively, when $\gamma = \text{may}$, only finite-time trajectories ending in a point y on the edge of the invariant such that at least a trajectory starting from y is cut out by the invariant. Similarly, when $\gamma = \text{must}$, only finite-time trajectories ending in a point y on the edge of the invariant such that all admissible trajectories from y are cut out by the invariant.

4.6.1 The brink Atomic Proposition

In order to reduce the two problems to the finite-time case of Section 4.4, we therefore need to restrict the set of trajectories considered in that problem to the correct one. To do this, we introduce a fresh atomic proposition brink, whose denotation depends on the semantics γ of interest and contains all and only the points x with the property discussed above.

$$[\mathit{brink}]^{\mathit{may}} \triangleq x \in \mathit{Inv} \wedge \exists f \in \mathit{Traj}_{\top}(x).\ f \text{ immediately enters } \mathit{Inv}$$

$$[\mathit{brink}]^{\text{must}} \triangleq x \in \mathit{Inv} \wedge \forall f \in \mathit{Traj}_{\top}(x).\ f \text{ immediately enters } \mathit{Inv}$$

The denotation of *brink* coincides with the polyhedron computed as follows:

$$[\mathit{brink}]^{\gamma} = \begin{cases} \mathit{reach}_0(\mathit{Inv}, \mathit{Inv}), & \text{if } \gamma = \mathit{may}; \\ \mathit{Inv} \setminus \mathit{reach}^{+}(\mathit{Inv}, \mathit{Inv}), & \text{if } \gamma = \text{must}. \end{cases}$$

Observe that, by the definitions of the operators reach_0 and reach^{+} in Section 2.5, the set $\mathit{reach}_0(\mathit{Inv}, \mathit{Inv})$ contains the points of the invariant from where an admissible trajectory immediately enters Inv. The set $\mathit{Inv} \setminus \mathit{reach}^{+}(\mathit{Inv}, \mathit{Inv})$, instead, contains those points x in the invariant from where no trajectory can remain in the invariant for a while. Therefore, if the system reaches any such point x, it can no longer move. Let $\mathcal{P}^{\mathit{brink}}_{\gamma}$ be the polyhedral system obtained from $\mathcal{P}$ by adding the proposition brink with denotation defined according to the semantics γ. We can prove the following theorem.

Theorem 7 ([**??**])**.** *For all* RTL_ω *formulae* φ *and semantics* $\gamma \in \{\mathit{may}, \mathit{must}\}$*, the following equivalence holds:*

$$\mathcal{P}, x \models^{\gamma} \varphi \wedge \mathsf{F}\,\mathit{last} \quad \mathit{iff} \quad \mathcal{P}^{\mathit{brink}}_{\gamma}, x \models^{\mathit{fin}} \varphi \wedge \mathsf{F}(\mathit{brink} \wedge \mathit{last})$$

The proof of the theorem can be found in [**??**]. The following corollary, which is an immediate consequence of the previous theorem, proves soundness and completeness of the decomposition.

Corollary 1 ([**??**]). *For all* RTL_ω *formulae* φ *and semantics* $\gamma \in \{may, must\}$*, the following equivalence holds:*

$$\mathcal{P}, x \models^{\gamma} \varphi \quad \textit{iff} \quad \mathcal{P}^{brink}_{\gamma}, x \models^{fin} \varphi \wedge \mathsf{F}(brink \wedge last) \;\; \textit{or} \;\; \mathcal{P}, x \models^{inf} \varphi$$

4.6.2 Omnidirectional Flow Scenario

According to Corollary 1, the model-checking problem for φ can be decomposed into the corresponding problem for the formula:

$$\varphi \wedge \mathsf{F}(brink \wedge last)$$

over the finite-time trajectories of the polyhedral system $\mathcal{P}^{brink}_{\gamma}$ and the problem of verifying φ over the infinite-time trajectories of $\mathcal{P}$. The first problem, in turn, can be solved by applying finite-time technique described in Section 4.4.2. The second one, instead, uses the tecnique described in Section 4.4.1.

Observe that we can avoid solving the subproblem on finite-time trajectories when $\gamma = \text{must}$, as it becomes trivial and always has a negative answer. Indeed, the set of trajectories $Traj_{\text{must}}$ coincides with $Traj_{\text{inf}}$, since any point $x \in Inv$ admits an infinite-time trajectory that stays in x forever. Hence, in this case, only the subproblem on infinite-time trajectories needs to be solved.

4.6.3 Forced Motion and Bounded Invariant Scenario

In this scenario, we assume that the origin lies outside the flow constraint ($\mathbf{0} \notin \text{cl}(Flow)$) and that the invariant is bounded. We call the first condition forced motion because the system cannot stay still, and cannot even slow down arbitrarily. Under these assumptions, no infinite-time trajectories exist (a proof can be found in [**??**]).

As a consequence of this and Corollary 1, we obtain the following.

Corollary 2. *For all* RTL_ω *formulae* φ *and semantics* $\gamma \in \{may, must\}$*, if* $\mathbf{0} \notin cl(Flow)$ *and* Inv *is bounded, then the following holds:*

$$\mathcal{P}, x \models_{\gamma} \varphi \quad \textit{iff} \quad \mathcal{P}_{brink}, x \models^{\mathsf{fin}} \varphi \wedge \mathsf{F}(brink \wedge last).$$

At this point, the model-checking problem can be solved as in Section 4.4.1.

4.6.4 Non-Recurrent `RTL` and Compact Flow Scenario

Similarly to the First Scenario in Section 4.6.2, the model-checking problem for φ is decomposed into the corresponding problem for the formula $\varphi \wedge \mathsf{F}(\mathit{brink} \wedge \mathit{last})$ over the finite-time trajectories of the polyhedral system $\mathcal{P}^{\gamma}_{\mathit{brink}}$ and the problem of verifying φ over the infinite-time trajectories of $\mathcal{P}$. The first problem is again solved using the technique presented in Section 4.4.1. The second one, instead, is addressed in Section 4.5 and, in turn, also reduces to the model checking of finite-time trajectories in the polyhedral system $\mathcal{P}_{\mathit{stay}}$.

Part II

Design & Implementation

—5—

Requirements and Analysis of the RTL Model Checking Tool

Contents: **5.1 Functional and Non-Functional Requirements.** 5.1.1 Functional Requirements Specification – 5.1.2 Non-Functional Requirements Specification. **5.2 High-level Architecture and Analysis Class Diagrams.** 5.2.1 High-Level Architecture – 5.2.2 Analysis Class Diagrams.

This chapter outlines the requirements and high-level architecture of the `RTL` Model Checking Tool. It defines what the system must achieve and introduces an abstract structural view of its components. Later chapters will refine this model into a concrete architecture and implementation.

5.1 Functional and Non-Functional Requirements

This section defines the functional and non-functional requirements of the `RTL` Model Checking Tool. The goal is to specify what the system must do and under which constraints it must operate. Requirements are grouped into two categories: functional requirements, describing the tool's capabilities, and non-functional requirements, covering implementation constraints, usability, maintainability, and performance.

5.1.1 Functional Requirements Specification

Table 5.1 summarises the functional requirements of the `RTL` Model Checking Tool. These requirements define the key functionalities that the tool must offer, including the parsing of input specifications, solving verification problems under different semantics, and exposing its features through a command-line interface.

Each requirement is assigned a *priority* value that indicates its relative importance for the system. A **M**ust have requirement is essential and mandatory for the tool's correct functioning. A **S**hould have requirement is important and highly recommended, but the system could still operate without it. A **C**ould have requirement is optional and may be implemented if time and resources allow, as it enhances the tool but is not critical. Although the table includes requirements with different priority levels, all listed features have been implemented in the final version of the tool.

Table 5.1: Functional Requirements of the `RTL` Model Checking Tool

ID	**Functional Requirement**	**Priority**
FR-01	The tool shall provide a command-line interface (CLI) exposing all core functionalities.	**M**
FR-02	The tool shall accept as input a formal specification of a polyhedral system and an `RTL` formula, parsing them and reporting syntactic or semantic errors clearly.	**M**
FR-03	The tool shall solve the existential denotation problem under all four semantics: fin, inf, may, and must.	**M**
FR-04	The tool shall solve the universal denotation problem under all four semantics: fin, inf, may, and must.	**M**
FR-05	The tool shall solve the model-checking problem at a rational point $\boldsymbol{p}$ for each semantics and both existential and universal modes.	**M**
FR-06	The tool shall automatically select the optimal resolution strategy for existential, universal, or model-checking problems based on the input semantics and characteristics.	**S**
FR-07	The tool shall include a statistics engine that reports relevant data such as execution time, number of states and transitions, and maximum patch count in denotations.	**S**
FR-08	The tool shall support exporting internal data structures (e.g., automata, polyhedral abstractions) as images or text suitable for graphical conversion.	**C**
FR-09	The tool shall provide configurable logging of all verification steps, supporting multiple verbosity levels to aid debugging and analysis.	**C**

5.1.2 Non-Functional Requirements Specification

Table 5.2 presents the non-functional requirements that guided the design and implementation of the system. These include constraints on implementation technologies, usability expectations, maintainability goals, and performance considerations.

Table 5.2: Non-Functional Requirements of the `RTL` Model Checking Tool

ID	**Non-Functional Requirement**
Implementation and Technology	
NFR-01	The tool shall be implemented in `C++17`.
NFR-02	The tool shall use the **Parma Polyhedra Library (PPL)** [20] for representing and manipulating polyhedra.
NFR-03	The tool shall use the **SPOT** [21] library for `RTL` formula parsing, automata construction, and model-checking algorithms.
NFR-04	The tool shall support exporting internal graph-based structures using the **DOT** format (Graphviz-compatible).
Usability	
NFR-05	The tool shall provide a clear and user-friendly CLI, including help messages and meaningful error reporting.
Maintainability and Testability	
NFR-06	The tool shall be modular and testable, with low coupling between components to facilitate unit and integration testing.
NFR-07	The tool shall follow `C++` best practices and be well-documented to ensure long-term maintainability.
Performance	
NFR-08	The tool should be optimised for performance, minimizing overhead when solving large instances of the model-checking problem.

5.2 High-level Architecture and Analysis Class Diagrams

This section presents a high-level architectural view of the `RTL` Model Checking Tool and provides an analysis-level class diagram for each major component. The goal is to outline the logical structure of the system before delving into implementation details.

5.2.1 High-Level Architecture

Figure 5.1 shows the high-level architecture of the `RTL` Model Checking Tool. The system is structured into modular components, each with a well-defined role. The `cli` module handles user interaction through a command-line interface. It collects the polyhedral system specification, the `RTL` formula, the desired semantics (fin, inf, may, or must), and additional configuration parameters (e.g., logging settings). After validating and sanitising the input, it passes it to the appropriate modules. The `parser` module processes the polyhedral system specification and `RTL` formula, building the internal data structures needed for verification. The `system` module provides a complete internal representation of the polyhedral system. The `discretisation` module converts $\mathtt{RTL}_\omega$ or $\mathtt{RTL}_\mathtt{f}$ into their discrete-time counterparts $\mathtt{LTL}_\omega$ or $\mathtt{LTL}_\mathtt{f}$. The `automata` module constructs Büchi or NFA automata from the discretised formulae, using classical LTL-to-automata translation techniques. It also supports computing the synchronous product between automata. The `abstractions` module provides polyhedral abstractions of the system, represented as automata. It interacts with both the `system` (to access the model) and `automata` (to support product operations between the abstraction and the formula automaton). The `mcsolvers` module implements the core verification algorithms. It solves existential and universal denotation problems for all supported semantics by coordinating the `abstractions`, `automata`, and `discretisation` modules.

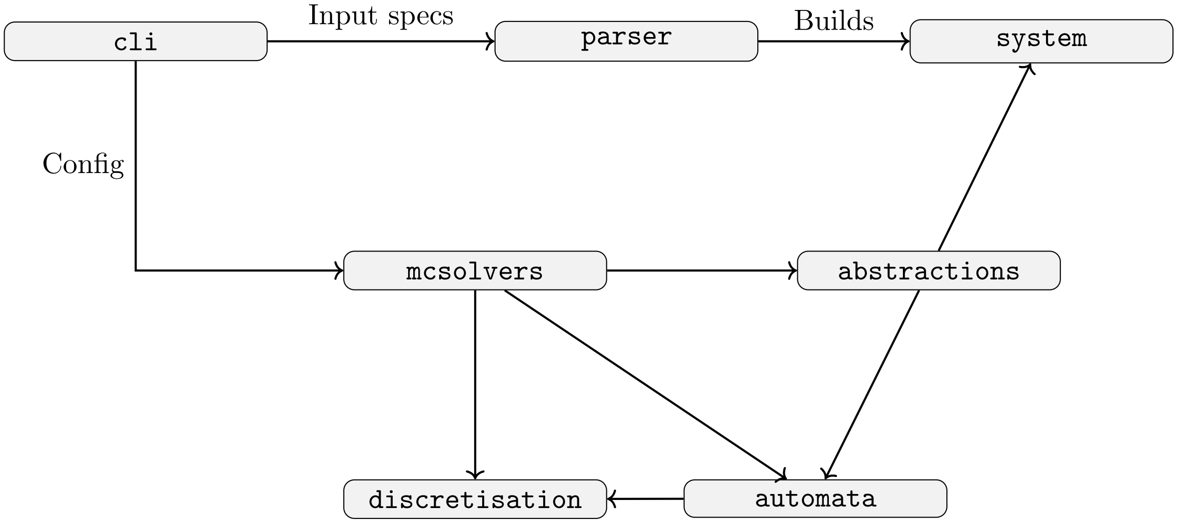


Figure 5.1: Components of the system and their interactions, illustrating the high-level architecture.

5.2.2 Analysis Class Diagrams

This section provides an analysis-level class diagram for each component defined in the high-level architecture (Figure 5.1). To facilitate the interpretation of these diagrams, we introduce a legend in Figure 5.2, which clarifies the visual conventions used:

- **C** indicates a concrete class;
- **A** indicates an abstract class;
- **E** indicates an enumeration.

Each class diagram follows these conventions consistently.

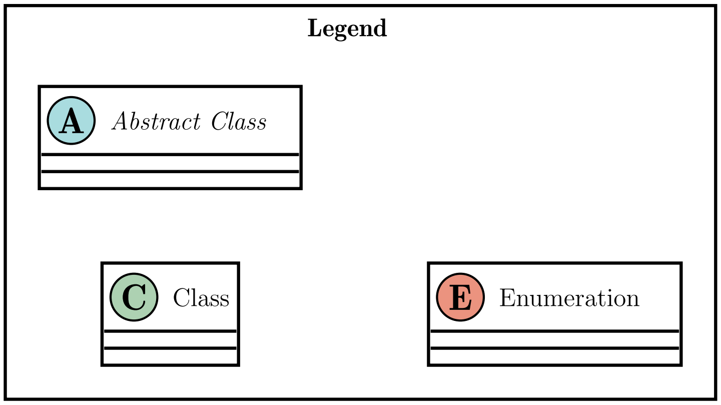


Figure 5.2: Visual legend for class diagrams.

Analysis Class Diagram for `cli`

The `cli` component is represented by the `RtlMcProgram` class (Figure 5.3), which stores all user-provided inputs required to run a verification task such as the system specification, `RTL` formula, selected semantics, mode (existential/universal), point for model checking, verbosity level, and output format. It is constructed from raw program arguments. Enumerations `Semantics`, `Verbosity`, and `OutputFormat` define the accepted values for key configuration parameters.

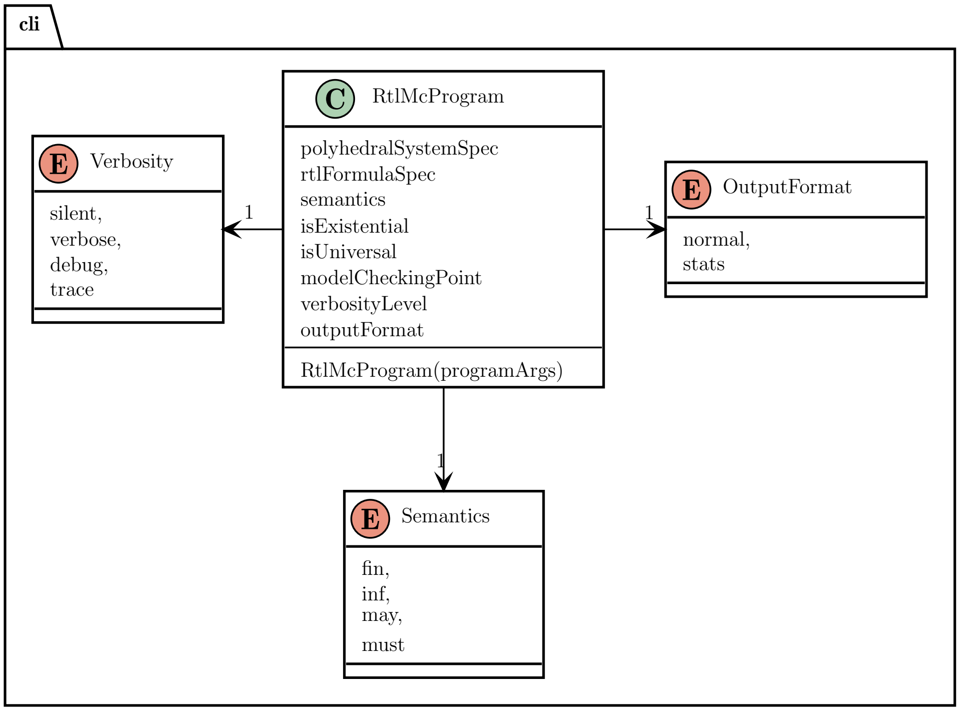

cli
RtlMcProgram
polyhedralSystemSpec
rtlFormulaSpec
semantics
isExistential
isUniversal
modelCheckingPoint
verbosityLevel
outputFormat
RtlMcProgram(programArgs)
Verbosity
silent,
verbose,
debug,
trace
OutputFormat
normal,
stats
Semantics
fin,
inf,
may,
must
1
1
1

Figure 5.3: `cli` Analysis Class Diagram

Analysis Class Diagram for `parser`

The `parser` component is represented by the `Parser` class (Figure 5.4), which provides two main methods: `parseRtlFormula`, which parses an `RTL` formula string into a `spot::formula` object (see Section 6.3.2), and `parseSystem`, which processes a string specification of a polyhedral system.

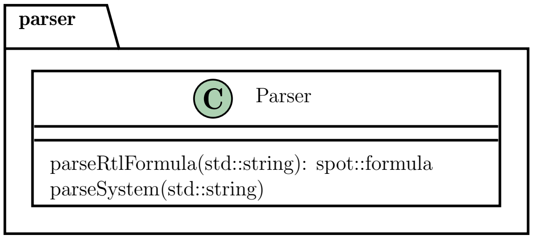

parser
Parser
parseRtlFormula(std::string): spot::formula
parseSystem(std::string)

Figure 5.4: `parser` Analysis Class Diagram

Analysis Class Diagram for `system`

The `system` component defines the internal representation of a polyhedral system and its observables. The main class `PolyhedralSystem` stores key attributes such as `flow`, `preFlow`, and `invariant`, and provides several analytical methods including `hasOmnidirectionalFlow()`, `isMovementForced()`, `hasCompactFlow()` and `interpretation(spot::formula)` to obtain the interpretation of an atomic proposition. The class `Observable` describes an observable (a set of atomic propositions together with its interpretation), while the class `AtomInterpretation` provides a mapping of atoms to their semantic meaning.

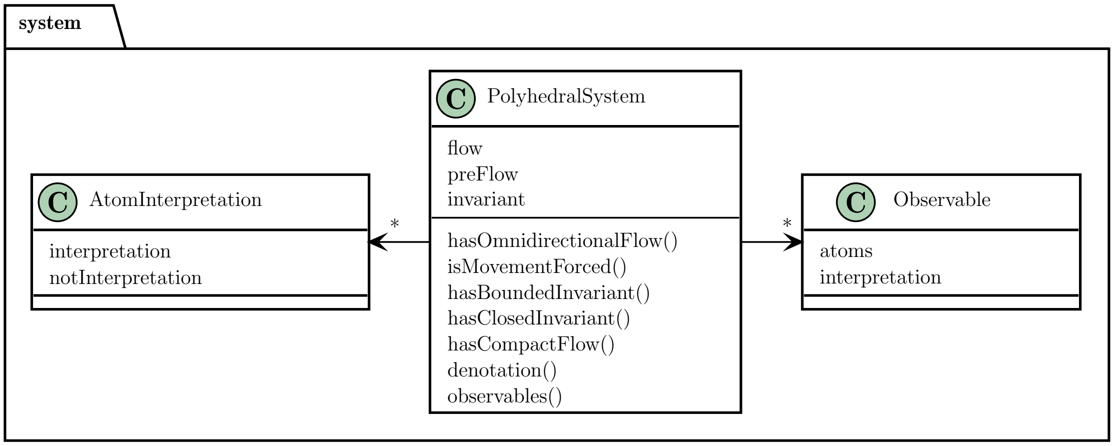

system
C PolyhedralSystem
flow
preFlow
invariant
hasOmnidirectionalFlow()
isMovementForced()
hasBoundedInvariant()
hasClosedInvariant()
hasCompactFlow()
denotation()
observables()
C AtomInterpretation
interpretation
notInterpretation
C Observable
atoms
interpretation
*
*

Figure 5.5: `system` Analysis Class Diagram

Analysis Class Diagram for `abstractions`

The class diagram for the `abstractions` module in Figure 5.6 defines the polyhedral abstractions. At the core of this design is the abstract class `PolyhedralAbstraction`, which encapsulates a `polyhedralSystem` and serves as the base for more specific abstraction strategies. One such implementation is `OmnidirectionalPolyhedralAbstraction`, which provides a concrete mechanism for constructing the abstraction by extracting `tileNodes` from the system. This class includes methods such as `extractTiles()`, which identify and group maximal connected regions, called tiles, associated with the same observable within the invariant, as discussed in Section 4.4.2. The class `Tile` models these connected regions.

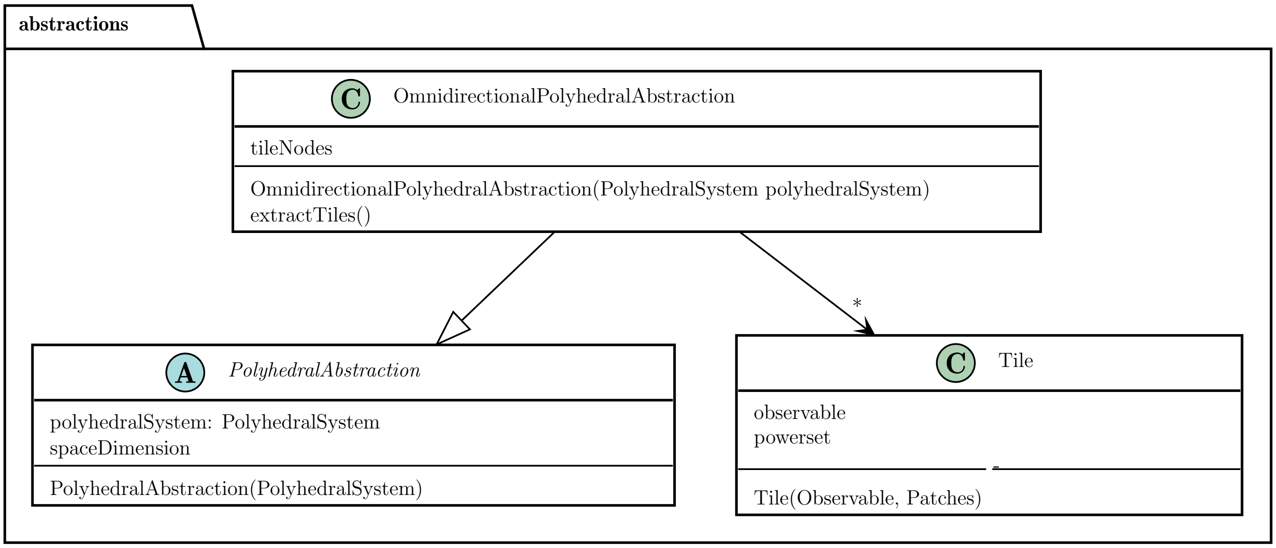

abstractions
C
OmnidirectionalPolyhedralAbstraction
tileNodes
OmnidirectionalPolyhedralAbstraction(PolyhedralSystem polyhedralSystem)
extractTiles()
*
A
PolyhedralAbstraction
polyhedralSystem: PolyhedralSystem
spaceDimension
PolyhedralAbstraction(PolyhedralSystem)
C
Tile
observable
powerset
Tile(Observable, Patches)

Figure 5.6: `abstractions` Analysis Class Diagram

Analysis Class Diagram for `discretisation`

The class diagram for `discretisation` in Figure 5.7 shows a single class, `DiscreteLtlFormula`, which represents a discrete linear temporal logic formula obtained from an `RTL` formula. It contains one attribute, `discreteLtlFormula` of type `spot::formula` (see Section 6.3.2), and a constructor that starts the discretisation process for an `RTL` formula represented as a `spot::formula` object.

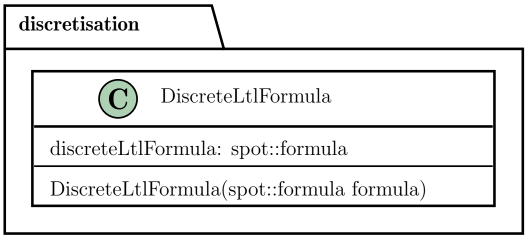

discretisation
C
DiscreteLtlFormula
discreteLtlFormula: spot::formula
DiscreteLtlFormula(spot::formula formula)

Figure 5.7: `discretisation` Analysis Class Diagram

Analysis Class Diagram for `automata`

The class diagram for `automata` in Figure 5.8 includes four main classes. The core abstract class is `Automaton`, which defines states, edges, and acceptance conditions. It provides several methods to query structural properties, such as `totalStates()`, `isInitialState(state)`, and `acceptance()`. The `AutomatonAcceptance` class models different acceptance types like Büchi and NFA. The `PolyhedralLtlAutomaton` class extends the automaton with a `DiscreteLtlFormula`, a polyhedral system and state denotations. This class represents the automaton corresponding to the LTL formula. Finally, the `PolyhedralSynchronousProductAutomaton` represents the synchronous product between an LTL automaton and a polyhedral abstraction.

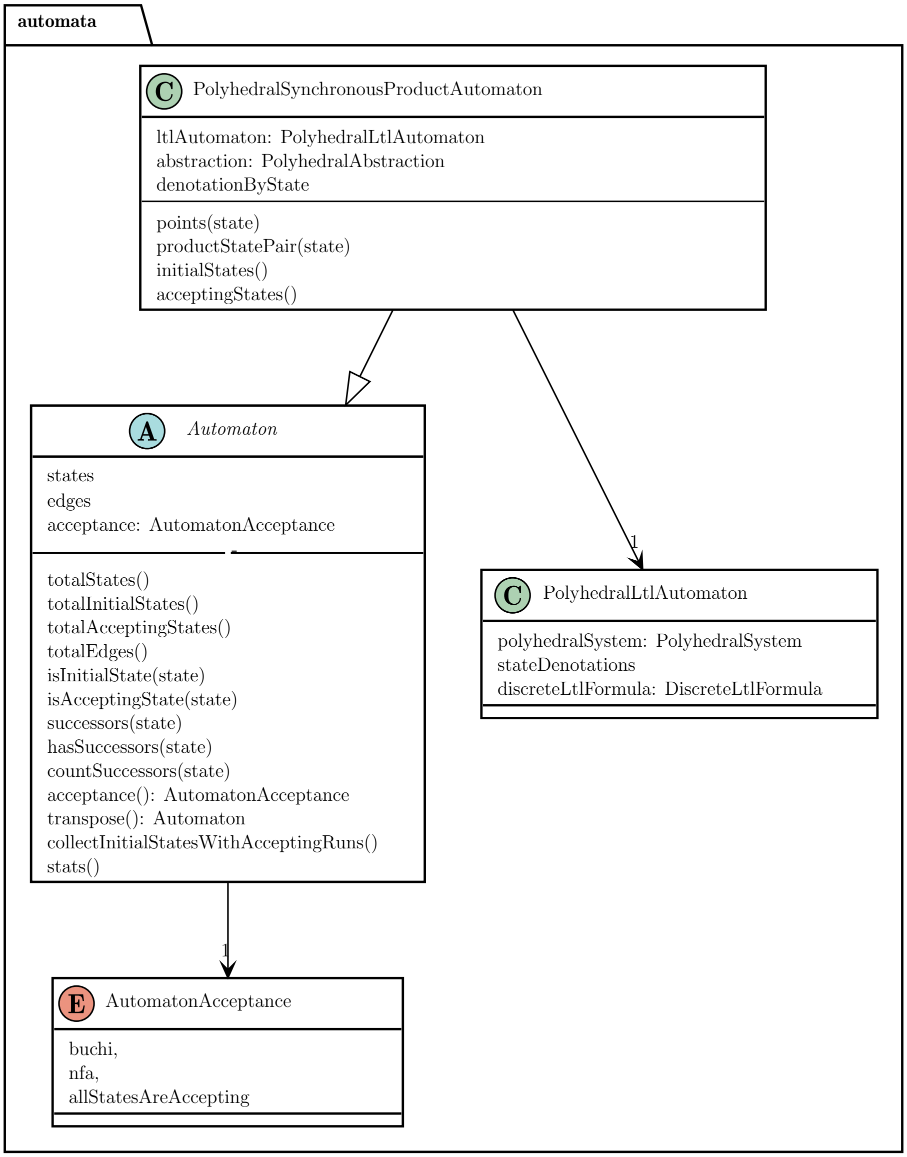

automata
C PolyhedralSynchronousProductAutomaton
ltlAutomaton: PolyhedralLtlAutomaton
abstraction: PolyhedralAbstraction
denotationByState
points(state)
productStatePair(state)
initialStates()
acceptingStates()
A Automaton
states
edges
acceptance: AutomatonAcceptance
totalStates()
totalInitialStates()
totalAcceptingStates()
totalEdges()
isInitialState(state)
isAcceptingState(state)
successors(state)
hasSuccessors(state)
countSuccessors(state)
acceptance(): AutomatonAcceptance
transpose(): Automaton
collectInitialStatesWithAcceptingRuns()
stats()
1
C PolyhedralLtlAutomaton
polyhedralSystem: PolyhedralSystem
stateDenotations
discreteLtlFormula: DiscreteLtlFormula
1
E AutomatonAcceptance
buchi,
nfa,
allStatesAreAccepting

Figure 5.8: `automata` Analysis Class Diagram

Analysis Class Diagram for `mcsolvers`

The `mcsolvers` class diagram in Figure 5.9 represents a collection of model checking algorithms designed to verify `RTL` properties over polyhedral systems. At the core of the design is the abstract class `Solver`, which encapsulates a reference to a `PolyhedralSystem`, along with the formula to be verified. This class defines a standard interface through key abstract methods, including:

- `solve()`, the main entry point for verification,
- `preprocessPolyhedralSystem()`, which prepares the system for analysis,
- `discretiseRtlFormula()` for translating the continuous-time logic into a discrete representation,
- `constructFormulaAutomaton()`, responsible for constructing the automaton representing the formula.

Concrete subclasses implement different verification strategies. The class `FiniteOnTheFlySolver` solves the problem of existential or universal denotation using the algorithm presented in Section 4.4.1 for the finite semantics. An extension of `FiniteOnTheFlySolver`, namely `MaximalOnTheFlySolver`, addresses the problem for the may or must semantics, distinguishing between the two via the `BrinkSemantics` enumeration, which assigns a semantics to the atomic proposition ***brink***. The abstract class `ExplicitSolver`, which extends `Solver`, holds a reference to the polyhedral abstraction and performs the synchronous product between the polyhedral abstraction and the formula automaton. Several concrete implementations of `ExplicitSolver` exist, all of which rely on the omnidirectional polyhedral abstraction and support different semantics (finite, infinite, and may).

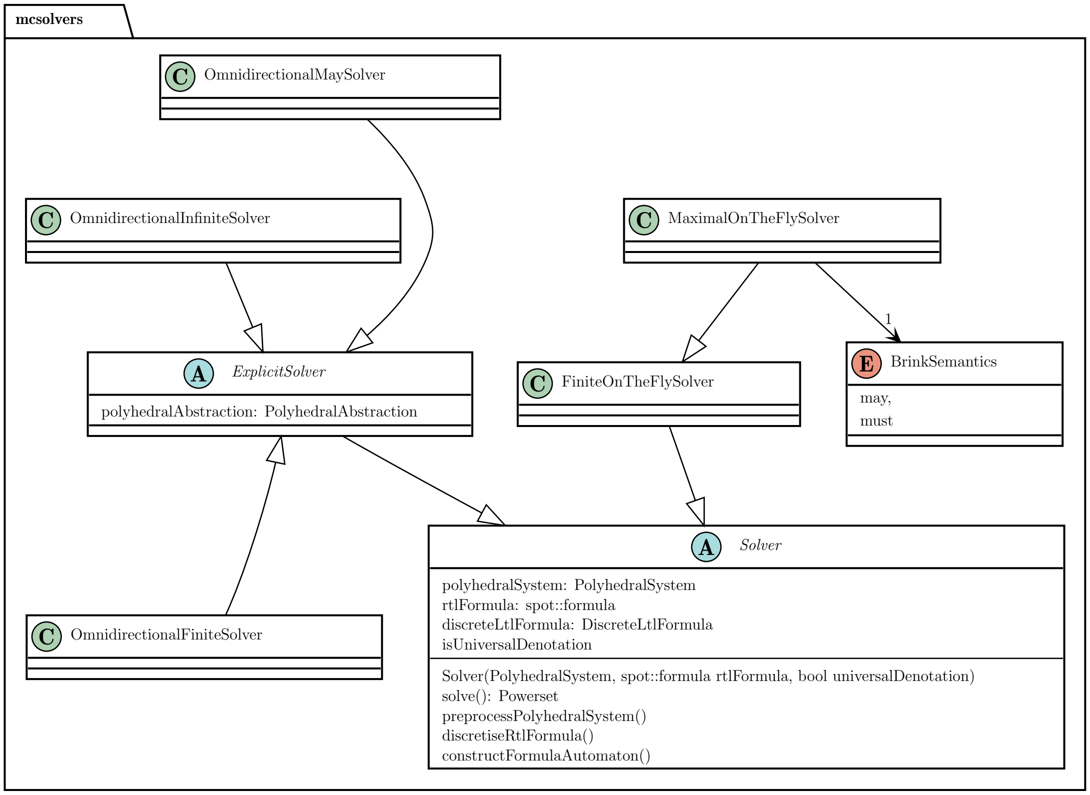

mcsolvers
C OmnidirectionalMaySolver
C OmnidirectionalInfiniteSolver
C MaximalOnTheFlySolver
A ExplicitSolver
polyhedralAbstraction: PolyhedralAbstraction
C FiniteOnTheFlySolver
1
E BrinkSemantics
may,
must
A Solver
polyhedralSystem: PolyhedralSystem
rtlFormula: spot::formula
discreteLtlFormula: DiscreteLtlFormula
isUniversalDenotation
Solver(PolyhedralSystem, spot::formula rtlFormula, bool universalDenotation)
solve(): Powerset
preprocessPolyhedralSystem()
discretiseRtlFormula()
constructFormulaAutomaton()
C OmnidirectionalFiniteSolver

Figure 5.9: `mcsolvers` Analysis Class Diagram

—6—

Project Structure and Fundamental Libraries

CONTENTS: **6.1 Project Structure and Dependencies.** 6.1.1 The Meson Build System – 6.1.2 Source Code Organisation – 6.1.3 Source Modules and Their Roles – 6.1.4 Project Dependencies. **6.2 Parma Polyhedra Library (PPL).** 6.2.1 Not Necessarily Closed Convex Polyhedron – 6.2.2 Double Description Method for Polyhedra – 6.2.3 Representing General Polyhedra – 6.2.4 Type Aliases for PPL Structures (`src/utils/ppl/ppl_aliases.h`) – 6.2.5 Main Operations on Polyhedra – 6.2.6 Utility Functions for PPL (`src/utils/ppl/ppl_utils.h`). **6.3 Spot.** 6.3.1 Spot Architecture Overview – 6.3.2 Constructing Formulae using `spot::formula` – 6.3.3 Parsing LTL Formulae – 6.3.4 Transition-based ω-Automata – 6.3.5 Binary Decision Diagrams – 6.3.6 Translation of LTL Formulae into Automata – 6.3.7 Working with $\texttt{LTL}_\text{f}$ Formulae with Finite-Time Semantics. **6.4 ANTLR4. 6.5 Grammars, Parsers, and Lexers. 6.6 Parse Tree Walking Mechanisms: Listeners and Visitors.**

This chapter describes the internal organisation of the `RTL` tool [26], focusing on the build system, source code structure, and key libraries that support its core functionality. We begin by outlining how the project is structured and compiled using the Meson build system. We then detail the main source components and external dependencies such as PPL and Spot, which enable symbolic manipulation of polyhedra and LTL-to-automata translation.

6.1 Project Structure and Dependencies

This section provides an overview of the build system used in the project, the organisation of the source code, the roles of the main components (packages), and the external dependencies required.

6.1.1 The Meson Build System

Meson [27] is a modern, efficient, and user-friendly build system designed to simplify the process of building software in an efficient manner. It uses a human-readable non-Turing complete domain-specific language (DSL) that allows users to describe the structure of their project in a clear, concise, and readable manner.

Meson Build Files

In the context of this project, Meson is used to define the build process for all components of the system. A key aspect of the Meson build system is the `meson.build` file. It defines how a specific part of the project should be built, specifying source files, dependencies, compiler options, and other configuration details. Written in Meson's custom build language, these files are used to describe the structure of the software and the necessary steps to compile and link the code. The root directory of the project contains a primary `meson.build` file, which sets the overall configuration, such as compiler flags, project version, and global options. In addition to the main build file, each subdirectory and submodule contains its own `meson.build` file. These files define how individual parts of the project are compiled and linked, and they are invoked recursively by the parent meson build files. This modular approach allows for a clear separation of concerns and makes the build system highly maintainable. In this project, Meson is also used to run scripts to generate ANTLR4 parsers automatically (see Section 6.4).

WrapDB and Dependency Management

Meson provides a system for managing external dependencies through the concept of *subprojects*. A subproject is an external Meson-based project nested within the `subprojects/` directory of the main project. This allows the main project to build and link the dependency directly. To simplify the inclusion of subprojects, Meson supports *Wrap files* (with extension `.wrap`). A Wrap file describes how to fetch and extract an external project automatically, removing the need to manually copy its source code. These Wrap files are often obtained from the *WrapDB*, an online repository maintained by the Meson project that hosts build definitions for numerous open-source libraries. When a Wrap file references a project available in WrapDB, Meson can automatically download and integrate it during the build process, streamlining dependency management. If a dependency cannot be handled as a subproject or via a Wrap file, it must be provided by the system. In such cases, Meson can locate it using tools like `pkg-config` [28], which helps Meson locate libraries installed on the system.

Configuration Options

Meson supports two kinds of configuration options:

- **Built-in options**: predefined by Meson itself, these control aspects such as the build type (e.g., `debug` vs. `release`), optimisation levels, warning settings, and installation paths.[1]

- **Build options**: Custom options defined by the project in the `meson_options.txt` file. These allow the user to enable or disable specific project features at configure time.

For this project, the following build options have been defined:

- `enable_tests`: Enables or disables the compilation of test suites.

- `generate_parser`: When enabled, automatically generates the ANTLR4 parser during the build process (see Section 6.4).

- `debug`: Enables debug-related features such as assertions.

Building and Running the Project with Meson

Meson requires a dedicated build directory to store all artifacts generated during the compilation process. This design allows developers to maintain multiple independent build directories with different configurations (e.g., with or without tests or debug mode), improving flexibility and speeding up development. The build directory should not be pushed to version control, as it is specific to the local environment.

Table 6.1: Common Meson commands

Command	**Description**
`meson setup buildDir`	Create a build directory
`meson setup buildDir -Doption=value`	Set build options at setup time
`meson configure -Doption=value buildDir`	Modify options in an build directory
`meson compile -C buildDir`	Compile the project
`meson compile --clean -C buildDir`	Clean and rebuild
`meson test -C buildDir`	Run all tests

6.1.2 Source Code Organisation

The project follows a well-organised directory structure, making it easy to navigate and manage. Below is the directory tree of the project:

---

[1] `https://mesonbuild.com/Builtin-options.html`

```
.
|--- CHANGELOG.md
|--- Dockerfile
|--- docs
|--- dot
|--- Doxyfile
|--- examples
|--- meson.build
|--- meson_options.txt
|--- README.md
|--- scripts
|--- src
|--- subprojects
|--- tests
|--- tools
```

The main directories and files are as follows:

- `CHANGELOG.md`: A file that records the history of changes made to the project.
- `Dockerfile`: The Dockerfile used to create a containerised version of the tool.
- `docs`: Contains documentation files for the project.
- `dot`: Folder related to Graphviz `.dot` files for visualisations.
- `Doxyfile`: Configuration file for Doxygen to generate project documentation.
- `examples`: Contains polyhedral system specification examples and `RTL` formula examples and usage scenarios for the project.
- `meson.build`: The main Meson build configuration file (see Section 6.1.1).
- `meson_options.txt`: File defining the options used in Meson to configure the build (see Section 6.1.1).
- `README.md`: A markdown file with essential information and instructions about the project.
- `scripts`: Contains essential scripts used to obtain the results of the experiments presented in Chapter 12.
- `src`: The main source code of the project (see Section 6.1.3).
- `subprojects`: Directory containing subprojects or dependencies managed by Meson (see Section 6.1.1).

- `tests`: Contains unit and integration tests for the project.
- `tools`: Contains additional tools (see Chapter 11).

6.1.3 Source Modules and Their Roles

The source code of the project is organised into modules located in the `src` directory, each encapsulating a specific functionality of the system. Below is a detailed description of each module and its role:

- `abstractions`: contains the abstract class `PolyhedralAbstraction` and its concrete implementations.
- `automata`: contains the abstract class `Automaton` and its implementations, including NFA and Büchi automata (see Section 4.3.3).
- `cli`: implements the `RtlMcProgram` class, which manages the command-line interface of the tool and handles user input.
- `discretisation`: contains the `DiscreteLtlFormula` class and functions that perform the discretisation of `RTL` formulae, following the transformations described in Section 4.3.
- `formula`: includes utility functions for manipulating logical formulae and constants of type `spot::formula` (see Section 6.3.2).
- `geometry`: provides functions for performing geometric operations on polyhedra, such as computing the border between two polyhedra or the interior of a polyhedron, as presented in Sections 2.1.2.
- `logger`: contains logging utilities used to trace the execution of key operations throughout the tool.
- `mcsolvers`: implements the hierarchy of model-checking solvers that realise the algorithms described in Chapter 4.
- `operators`: contains implementations of core operators used in the analysis, including the reachability operators (see Section 2.5).
- `parser`: provides the parser logic for reading `PolyhedralSystem` specifications and `RTL` formulae from input files (see Section 6.4).
- `product`: implements the `PolyhedralSynchronousProductAutomaton` class, which constructs the synchronous product between a polyhedral abstraction and the formula automaton.
- `stats`: contains all classes used to implement the statistics engine presented in Section 11.4, used to collect and report data from experiments.

- `system`: includes the `PolyhedralSystem` class and its components, which represent the input system under verification.
- `utils`: provides general-purpose utility functions, including helpers for the Parma Polyhedra Library (PPL, see Section 6.2) and the Spot library (see Section 6.3).

6.1.4 Project Dependencies

The project relies on a set of libraries to support its core functionalities, including symbolic manipulation of polyhedra, parsing, automata-based model checking, and logging. The Meson build system is used to declare and manage these dependencies, ensuring version compatibility and reproducibility across environments. The full list of dependencies, as declared in the project's `meson.build` files, is as follows:

- **PPL** (`ppl`): version `1.2`, used for representing and manipulating convex and general polyhedra (see Section 6.2).

  ```
  ppl_dep = dependency('ppl', version: '==1.2')
  ```

- **GMP** and **GMPXX** (`gmp`, `gmpxx`): versions `>=6.1.0`, required as back-end libraries for numerical operations and rational arithmetic.

  ```
  gmp_dep = dependency('gmp', version: '>=6.1.0')
  gmpxx_dep = dependency('gmpxx', version: '>=6.1.0')
  ```

- **ANTLR4 Runtime** (`antlr4-runtime`): version `>=4.13.0`, provides runtime support for grammars generated with ANTLR4 (see Section 6.4), used for parsing both system and the `RTL` formula.

  ```
  antlr4_runtime_dep = dependency(
      'antlr4-runtime',
      version: '>=4.13.0'
  )
  ```

- **Spot** (`libspot`): version `>=2.12.1`, used for parsing and translating LTL formulae into automata, as well as for automata manipulations (see Section 6.3).

  ```
  spot_dep = dependency('libspot', version: '>=2.12.1')
  ```

- **argparse** (from WrapDB): version `>=3.1`, lightweight C++ argument parser for CLI input handling.

  ```
  argparse_dep = dependency(
      'argparse',
      fallback : ['argparse'],
      version: '>=3.1'
  )
  ```

- **spdlog** (from WrapDB): version `>=1.15.0`, fast and header-only logging library with support for custom formatting and sinks.

  ```
  spdlog_dep = dependency(
      'spdlog',
      fallback : ['spdlog'],
      version: '>=1.15.0'
  )
  ```

- **fmt** (from WrapDB): version `>=11.1.0`, formatting library used by `spdlog` and directly in several components of the tool for string formatting.

  ```
  fmt_dep = dependency('fmt', fallback: ['fmt'], version: '>=11.1.0')
  ```

The dependencies `argparse`, `spdlog`, and `fmt` are retrieved from WrapDB (see Section 6.1.1), a package registry used by Meson to automatically fetch and build external projects. All other libraries are expected to be available as system-wide installations or to be provided by the user.

6.2 Parma Polyhedra Library (PPL)

The *Parma Polyhedra Library* (PPL) [20] is a `C++` library for the symbolic manipulation of numerical abstractions over $\mathbb{Q}^n$, including convex polyhedra (both closed and not necessarily closed), bounded difference shapes, grids, and more. These abstractions are widely used in static analysis, hybrid system modelling, and formal verification. Within the `rtl-mc` tool, PPL is employed to represent the system's continuous state space symbolically as convex polyhedra. The library provides core operations such as:

- Intersection, union, and difference of polyhedra;

- Affine transformations (image and preimage);
- Topological closure and dimension manipulation;
- Positive time-elapse under differential inclusions [29, 9].

PPL supports both constraint and generator representations, and internally uses the double description method (see Section 6.2.2) to switch between them efficiently.

6.2.1 Not Necessarily Closed Convex Polyhedron

The *Not Necessarily Closed* (NNC) convex polyhedron is the primary geometric domain used in the `rtl-mc` tool. In the PPL, an NNC polyhedron is defined as the intersection of a finite set of open or closed affine half-spaces in $\mathbb{Q}^n$, allowing the representation of strict inequalities. NNC polyhedra are internally handled using both constraint and generator representations, and support all core operations including intersection, affine image, and time-elapse.

6.2.2 Double Description Method for Polyhedra

The *double description method* is a fundamental technique used in the PPL to support dual representations of convex polyhedra:

- the *constraint representation*, in which a polyhedron is defined by a system of linear (in)equalities;
- the *generator representation*, in which it is described by a finite set of points, rays, and lines.

Switching between these two views enables efficient implementation of different operations: for example, intersections and inclusion tests are more naturally handled in constraint form, while convex hulls and Minkowski sums benefit from the generator form. Figure 6.1 illustrates a simple polyhedron defined by a set of linear constraints (top-right) and equivalently described using its generators (bottom-right). This example highlights how the same object can be analysed and manipulated using either representation depending on the algorithmic context.

6.2.3 Representing General Polyhedra

The PPL provides a *finite powerset construction* that lifts a base domain, such as convex polyhedra, to a disjunctive domain capable of representing arbitrary (i.e., non-convex) polyhedra as finite unions of convex ones. Internally, the `Pointset_Powerset` class is used to store and manipulate such collections, supporting standard operations like intersection, union, widening, and affine transformations on each disjunct.

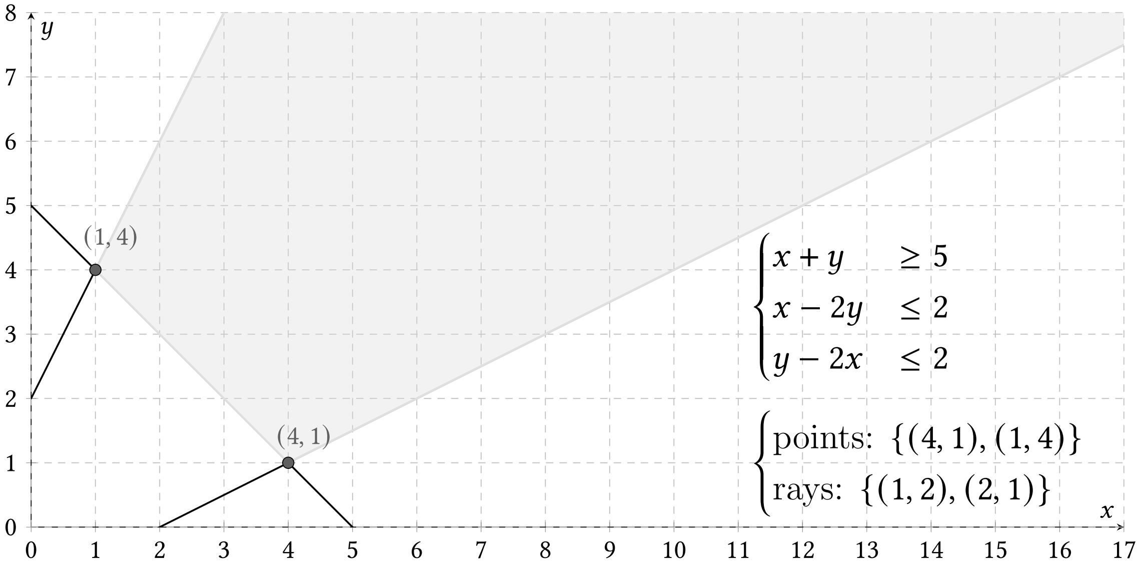


Figure 6.1: Example of a convex polyhedron represented both as a system of linear constraints (top-right) and as a generator system (bottom-right), consisting of two points and two rays.

6.2.4 Type Aliases for PPL Structures (`src/utils/ppl/ppl_aliases.h`)

To improve code readability, the header file `ppl_aliases.h` in the `utils` component defines a set of type aliases for the most common PPL types used in the project. The Table 6.2 summarises them:

Alias	**Description**
`Poly`	Alias for `PPL::NNC_Polyhedron`. Represents a convex polyhedron with strict inequalities.
`PolyUniquePtr`	`std::unique_ptr<Poly>`. Manages exclusive ownership of a dynamic polyhedron.
`PolySharedPtr`	`std::shared_ptr<Poly>`. Allows shared ownership of a modifiable polyhedron.
`PolyConstSharedPtr`	`std::shared_ptr<const Poly>`. Shared ownership of a constant polyhedron.
`PolyConstUniquePtr`	`std::unique_ptr<const Poly>`. Unique ownership of a constant polyhedron.
`Powerset`	`PPL::Pointset_Powerset<Poly>`. Represents a finite union of convex polyhedra.
`PowersetUniquePtr`	`std::unique_ptr<Powerset>`. Unique ownership of a polyhedron set.
`PowersetSharedPtr`	`std::shared_ptr<Powerset>`. Shared ownership of a polyhedron set.
`PowersetConstSharedPtr`	`std::shared_ptr<const Powerset>`. Shared ownership of a constant polyhedron set.
`PowersetConstUniquePtr`	`std::unique_ptr<const Powerset>`. Unique ownership of a constant polyhedron set.
`PPL`	Alias for `Parma_Polyhedra_Library`. Used to shorten namespace references.

Table 6.2: Type aliases defined in `ppl_aliases.h` for core PPL types.

6.2.5 Main Operations on Polyhedra

The Parma Polyhedra Library provides a wide range of operations for manipulating polyhedra. These include Boolean operations, affine transformations, and dimension management. Table 6.3 summarises the most relevant operations used in the `RTL` Model Checking Tool for the `Poly` type (i.e. individual convex polyhedra) and their corresponding PPL methods. In addition to convex polyhedra, the tool extensively uses the `Powerset` type to represent general (possibly non-convex) sets as finite unions of `Poly`.

Operation on `Poly`	**PPL Method**
Intersection	`intersection_assign(Poly)`
Union (hull)	`upper_bound_assign(Poly)`
Difference	`difference_assign(Poly)`
Affine image	`affine_image(Var, Expr)`
Affine preimage	`affine_preimage(Var, Expr)`
Closure	`topological_closure()`
Time-elapse	`positive_time_elapse_assign(Poly)`
Add dimensions	`add_space_dimensions_and_embed(n)`
Project dimensions	`add_space_dimensions_and_project(n)`
Remove dimensions	`remove_space_dimensions(Set)`
Remove higher dims	`remove_higher_space_dimensions(d)`
Is empty	`is_empty()`
Is universe	`is_universe()`

Table 6.3: Main operations on single convex polyhedra (`Poly`) and their PPL methods.

6.2.6 Utility Functions for PPL (`src/utils/ppl/ppl_utils.h`)

The Parma Polyhedra Library (PPL) provides an extensive set of operations for manipulating polyhedra, primarily designed around mutating methods that modify objects in-place (e.g., methods with the `_assign` suffix). While these operations are efficient, they often introduce boilerplate code and can complicate the flow of logic, especially when temporary objects are involved or when it is undesirable to modify objects in-place. To address these limitations and improve usability, the `ppl_utils.h` module introduces a set of utility functions that wrap around the core PPL operations. These functions follow a more functional programming style, ensuring that objects are not modified in-place, but instead new instances are returned. Additionally, the utility functions leverage *move semantics*, implemented in PPL via `swap` operations, to allow efficient transfer of ownership without unnecessary copies. For example, consider the following comparison between the raw PPL API and the utility wrapper provided by `ppl_utils.h`:

Code 6.1 Raw PPL API vs PPL Utility Wrapper

```
Poly a = ...;
Poly b = ...;

// Using raw PPL API
Poly result { a };
result.intersection_assign { b };

// Using PPLUtils
```

```
PolyUniquePtr result { PPLUtils::intersect(a, b) };
```

In the raw PPL API, the intersection operation modifies the existing object `result` in-place. This can result in cumbersome code when chaining operations. In contrast, the utility function returns a new object, ensuring that the original objects remain unchanged, and ownership semantics are handled using smart pointers like `PolyUniquePtr`. Moreover, the `PPLUtils` namespace provides functions for:

- Creating `Poly`hedra and `Powerset`s from constraints or initialiser lists. This feature is heavily used throughout the testing infrastructure of the tool, where symbolic test inputs are programmatically constructed to validate the correctness of geometric operations, observable generation, and model-checking procedures.
- Intersecting, merging, and performing set operations (union, difference) on `Poly`hedra and `Powerset`s.
- Converting `Poly`hedra and `Powerset`s to human-readable string representations.
- Efficiently handling memory management with move semantics.

This additional abstraction layer enhances code clarity, reduces the risk of bugs related to in-place modification, and minimises the amount of boilerplate code required for working with PPL structures. The Table 6.4 summarises some of the most useful utility functions available in `ppl_utils.h`:

This utility layer provides an abstraction over the PPL API, significantly improving code readability and maintainability by avoiding in-place modifications and manual memory management. In some cases, it can also improve performance by reducing unnecessary copies and leveraging move semantics.

`std::string toString(const Poly& poly)` *Converts a Poly object to a human-readable string.*
`std::string toString(const Powerset& powerset)` *Converts a Powerset object to a string.*
`Poly poly(std::vector<Constraint>&& constraints)` *Creates a Poly from a vector of constraints.*
`Poly poly(std::vector<Constraint>&& constraints, dimension_type dim)` *Creates a Poly with a specified dimension.*
`Powerset powerset({{...}}, dimension_type dim)` *Constructs a Powerset from a list of Poly constraints.*
`PolyUniquePtr intersect(const Poly& a, const Poly& b)` *Computes the intersection of two Poly objects.*
`PowersetUniquePtr intersect(const Powerset& a, const Powerset& b)` *Computes the intersection of two Powerset objects.*
`PowersetUniquePtr fusion(const Powerset& a, const Powerset& b)` *Returns the union (fusion) of two Powersets.*
`PowersetUniquePtr minus(const Powerset& a, const Powerset& b)` *Returns the set difference a - b.*
`PowersetUniquePtr complement(const Powerset& a)` *Computes the complement of the given Powerset.*
`Poly point(Linear_Expression expr)` *Creates a Poly representing a point.*
`Poly zeroPoint(dimension_type dim)` *Creates a Poly representing the origin.*
`bool containsDisjunct(const Powerset& ps, const Poly& p)` *Checks if a Powerset contains a given Poly.*
`bool isSingleVariableZeroEqualityConstraint(const Constraint& c)` *Checks if a constraint is of the form* $x = 0$.
`PolyUniquePtr removeSingleVariableZeroEqualityConstraints(const Poly& poly)` *Removes* $x = 0$ *constraints from the Poly.*

Table 6.4: Utility functions in `PPLUtils` with their signatures and descriptions.

6.3 Spot

Spot [21] is a `C++17` library for LTL, ω-automata manipulation and model checking. It supports multiple syntaxes for LTL, arbitrary acceptance conditions for ω-automata, and offers a range of algorithms for formula manipulation, including translation to automata. In the context of this tool, Spot is primarily used to represent `RTL` formulae and to translate these formulae into automata, enabling the execution of model checking algorithms.

6.3.1 Spot Architecture Overview

The Spot project consists of various components, but for this tool, we focus on two key libraries: `libspot` and `libbddx`. `libspot` is the core `C++17` library, providing the essential data structures and algorithms for manipulating LTL formulae and performing model checking. It is responsible for representing and handling LTL formulae and translating them into automata used for verifying temporal properties. `libbddx` is a customised version of the BuDDy library [30], designed for manipulating Binary Decision Diagrams (BDDs, see Section 6.3.5). Together, `libspot` and `libbddx` form the foundation of the tool's functionality. As shown in Figure 6.2, the architecture of the Spot project is quite large and consists of several parts, but only the `libspot` and `libbddx` libraries are relevant to this tool.

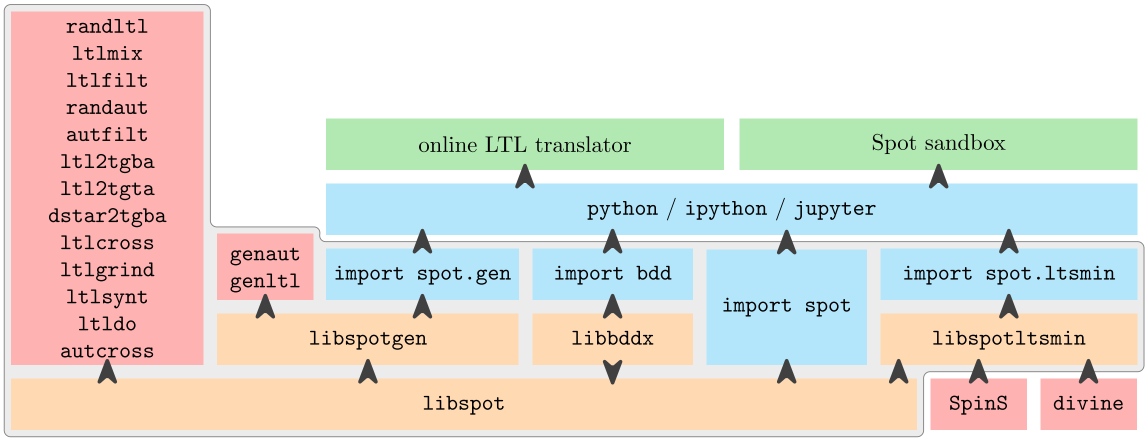

randltl
ltlmix
ltlfilt
randaut
autfilt
ltl2tgba
ltl2tgta
dstar2tgba
ltlcross
ltlgrind
ltlsynt
ltldo
autcross
online LTL translator
Spot sandbox
python / ipython / jupyter
genaut
genltl
import spot.gen
import bdd
import spot
import spot.ltsmin
libspotgen
libbddx
libspotltsmin
libspot
SpinS
divine

Figure 6.2: Architecture of the Spot Project

6.3.2 Constructing Formulae using `spot::formula`

The primary class for constructing temporal formulae in Spot is `spot::formula`. This class provides static methods that act as constructors for each supported operator (factory methods). Boolean constants can be created using `spot::formula::tt()` for $\top$ and `spot::formula::ff()` for $\bot$. Atomic propositions are built with `spot::formula::ap("name")`. Unary and binary operators follow straightforward syntax such as `spot::formula::F(arg)` for the `F` (eventually) operator and `spot::formula::U(first, second)` for the `U` (until) operator. For n-ary operators, like $\wedge$, an initialiser list is used: `spot::formula::And(arg1, arg2, arg3)`. The following code snippet demonstrates how to build a simple LTL formula representing $\mathtt{F}(p \wedge \mathtt{F}(q))$:

Code 6.2 Spot formula example

```
#include <spot/tl/formula.hh>

int main()
{
    spot::formula f {
        spot::formula::F(
            spot::formula::And({
                spot::formula::ap("p"),
                spot::formula::F(spot::formula::ap("q"))
            })
        )
    };
}
```

Although one could use `using namespace spot` to simplify the syntax, this only removes the need to qualify the `spot` namespace. The `formula::` prefix, however, must still be explicitly written for each factory method, which can reduce readability in deeply nested expressions. To address this and improve syntactic clarity, the `formula` component of the tool defines a set of wrapper functions that forward calls to the corresponding `spot::formula::` factory methods. This allows writing LTL expressions more succinctly, avoiding repetitive qualification and improving code readability throughout the tool. The structure of the formula is shown in Figure 6.3, where each node in the tree represents a formula component.

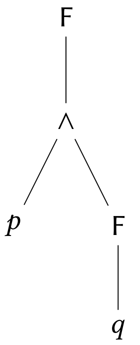


Figure 6.3: Tree representation of the LTL formula $\mathsf{F}(p \land \mathsf{F}(q))$ constructed using `spot::formula`.

The `spot::formula` class allows easy manipulation of such structures, supporting operations like traversal and transformation. For instance, the `traverse` method enables recursive exploration of subformulae, while `map` applies a transformation to all subformulae, making it easier to process and manipulate complex LTL formulae. The formula is explored in a DFS fashion (without skipping subformulae that appear twice).

6.3.3 Parsing LTL Formulae

Spot provides several methods to parse LTL formulae. This tool focuses on the `parse_infix_psl()` method, which uses infix notation for parsing LTL formulae. Infix notation is a standard mathematical expression format where operators are placed between operands (e.g., $a \wedge b$). The `parse_infix_psl()` function builds a formula from an LTL string and returns a `parsed_formula` object. To ensure that parsing was successful, the `parsed_formula::errors` field should be checked for any errors. In Spot, LTL formulae can be written using various syntaxes supported by different tools, such as Spin, Wring, and Goal. The `parse_infix_psl()` function can parse all of these syntaxes. Spot's default parser also supports the linear fragment of PSL (Property Specification Language), which extends LTL with regular expressions. However, this tool focuses solely on LTL, and any attempt by the user to use this extended syntax will be detected by the tool. In Spot, atomic propositions are any alphanumeric strings that do not start with the uppercase characters F, G, or X. For example, `gfa` is an atomic proposition, while `GFa` denotes the LTL formula $\mathsf{G}(\mathsf{F}(a))$. Additionally, any double-quoted string is considered an atomic proposition, allowing complex expressions to be treated as atomic propositions that Spot will not attempt to interpret. Spot's parser supports multiple syntaxes to express the same LTL semantics. Tables 6.5 and 6.6 highlight the preferred syntax for Spot's for logical and temporal operators. The parser also supports syntactic sugar for certain operators. For more details on the syntax and supported operators, refer to [31]. The following code in Listing 6.3 demonstrates how to parse the formula $\mathsf{F}(p \wedge \mathsf{F}(q))$ using `parse_infix_psl()`:

Code 6.3 Parsing LTL Formula Example

```
#include <spot/tl/parse.hh>
#include <spot/tl/print.hh>

int main()
{
    spot::parsed_formula pf { spot::parse_infix_psl("F(p & F(q))") };
    if (pf.format_errors(std::cerr))
        return 1;
    std::cout << pf.f << '\n';
    return 0;
}
```

The code in Listing 6.3 parses an LTL formula in infix notation, checking for errors, and prints the resulting formula.

Logical Operators

Operation	**Preferred Syntax**
Negation	`!`f
Disjunction	$f \mid g$
Conjunction	f & g
Implication	f `->` g
Exclusion	f `xor` g
Equivalence	f `<->` g

Table 6.5: Logical Operators Syntax

Temporal Operators

Operator	**Preferred Syntax**
(Weak) Next	`X`f
Strong Next	`X[!]`f
Eventually	`F`f
Always	`G`f
(Strong) Until	f`U`g
Weak Until	f`W`g
(Weak) Release	f`R`g
Strong Release	f`M`g

Table 6.6: Temporal Operators Syntax

6.3.4 Transition-based ω-Automata

In Spot, the primary class for representing ω-automata is `spot::twa`, which stands for *Transition-based ω-Automaton* (TωA). These automata are labelled by boolean formulae, represented using Binary Decision Diagrams (BDDs), as discussed in Section 6.3.5. An ω-word is accepted by a TωA if there exists an accepting run where the labels of the visited edges (the boolean formulae) are compatible with the minterms corresponding to the symbols in the word. In this framework, the automaton's transitions are labelled not with individual letters, but with boolean formulae, which can be viewed as a way of aggregating several letter-labelled transitions into one. TωAs are a specific type of transition-based ω-automaton. This means that, in addition to the usual labels on transitions, the automaton includes an acceptance condition defined in terms of sets of transitions (or edges, depending on your terminology). In transition-based acceptance, a run is accepted if the set of edges it visits satisfies the acceptance condition. Figure 6.4 illustrates an example of a Transition-based Büchi Automaton (TBA), where accepting transitions are highlighted in green. In this example, the automaton accepts all ω-words where the pattern $a^+;b$ is repeated infinitely often: that is, a positive number of a followed by one occurrence of b. Using transition-based acceptance often leads to more compact automata. For instance, the automaton shown in Figure 6.4 would require at least three states if using state-based acceptance, as shown in Figure 6.5. Internally, instead of representing acceptance sets as actual sets of edges, Spot labels each edge with a bit-vector that lists the acceptance sets it belongs to. Each automaton has a flag indicating whether it uses state-based or transition-based acceptance. Regardless of the flag's value, membership in acceptance sets is always stored on transitions. In automata with state-based

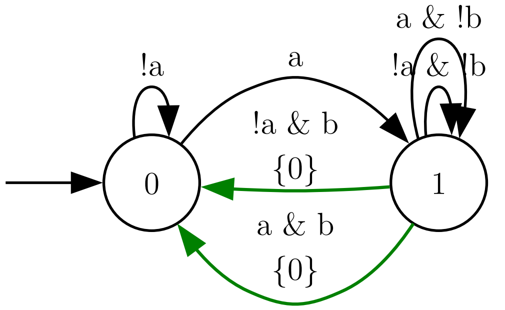


Figure 6.4: Transition-Based ω-Automaton

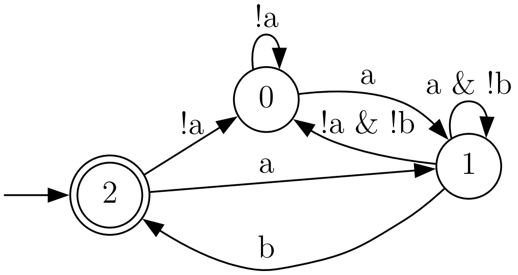


Figure 6.5: State-Based ω-Automaton

acceptance, all transitions leaving a state carry the acceptance-set membership of that state. This allows an automaton with state-based acceptance to be interpreted as one with transition-based acceptance when needed. In the special case where an accepting state has no outgoing transitions, a situation that may arise, for instance, when building a Non-deterministic Finite Automaton (NFA, see Section 6.3.7), it is necessary to artificially create an outgoing transition marked as accepting. Spot automatically handles this by inserting a self-loop labelled with the constant false (i.e. `0`) and marked as accepting. This ensures that the semantics of acceptance are preserved even when the state lacks outgoing edges.

6.3.5 Binary Decision Diagrams

Spot use Binary Decision Diagrams (BDDs) for various purposes, with the primary use being the labelling of automaton edges. Each edge stores a BDD representing its guard, which is a boolean function over atomic propositions. While the automaton is represented as a graph (with vectors of states and edges), the BDD is used exclusively for the guard, distinguishing it from symbolic representations where the entire transition structure is encoded as a single large BDD. From the perspective of the BDD library [30], BDDs are directed acyclic graphs (DAGs) with nodes labelled by BDD variables (numbered from 0). In Spot's algorithms, these variables have specific meanings. For instance, when synchronising two automata with guards over atomic propositions such as a and b, it is crucial that both automata agree on the BDD variables representing these propositions. The `spot::bdd_dict` object manages the allocation of BDD variables and ensures that variables are reused consistently across different automata when they serve similar purposes. For example, when performing operations like `spot::product(aut1, aut2)` to compute the synchronous product of two automata, it is required that `aut1->get_dict()` equals `aut2->get_dict()`. When constructing a `spot::twa_graph` automaton (an implementation of `spot::twa`) a `bdd_dict` is passed as an argument to manage the BDD variables.

6.3.6 Translation of LTL Formulae into Automata

Spot provides the ability to translate any LTL or PSL formula into Büchi or generalized Büchi automata. Internally, the translation process generates Transition-based Generalized Büchi Automata (TGBA), which can later be simplified using various algorithms based on the selected options. The entire translation pipeline, ranging from simplifying the formula, converting the simplified formula into an automaton, and optimising the resulting automaton, is managed by the `spot::translator` class. This class can be configured using the following methods:

- `set_type()` to choose the desired type of automaton (e.g. Büchi, generalized Büchi),
- `set_level()` to control the optimisation level (set to high by default),
- `set_pref()` to set preferences like small size or determinism, and characteristics like completeness, unambiguity, or state-based acceptance.

Figure 6.6 shows the automaton resulting from the translation of the LTL formula $F(p \wedge F(q))$.

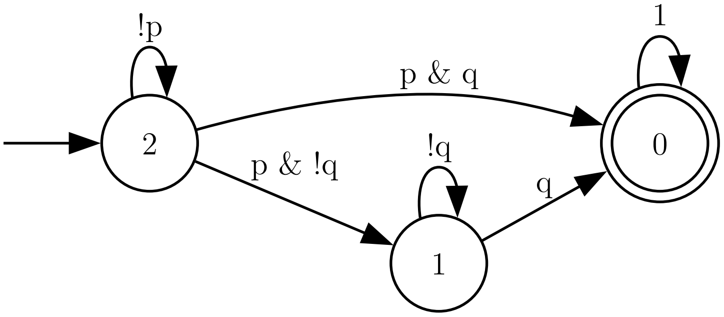


Figure 6.6: Automaton resulting from the translation of the LTL formula $F(p \wedge F(q))$.

6.3.7 Working with LTL$_\mathrm{f}$ Formulae with Finite-Time Semantics

Spot is primarily designed to work with LTL$_\omega$ operators over infinite words, with all supported automata being ω-automata (i.e. automata over infinite words). However, Spot can also handle LTL$_\mathrm{f}$ (LTL with finite semantics) properties by transforming the finite semantics into infinite semantics using a specific technique. The procedure is as follows:

1. **Input LTL$_\mathrm{f}$ Formula**: Spot first reads the LTL$_\mathrm{f}$ formula as if it were a standard LTL$_\omega$ formula (see Section 6.3.3).

2. **Rewrite to Embed $\mathtt{LTL_f}$ Semantics**: The formula is then modified to embed the semantics of $\mathtt{LTL_f}$ within the $\mathtt{LTL_\omega}$ framework. A new atomic proposition *alive* is introduced. This proposition is initially true but eventually becomes false permanently. All `LTL` operators are adjusted to require satisfaction only during the *alive* part of the word. For example, the formula $(a \,\mathsf{U}\, b) \wedge \mathsf{F}(c)$ is transformed into $alive \wedge (a \,\mathsf{U}\, (alive \wedge b)) \wedge \mathsf{F}(alive \wedge c) \wedge (alive \,\mathsf{U}\, \mathsf{G}\neg alive)$.

3. **Convert to Büchi Automaton**: The modified formula is then converted into a Büchi automaton (see Section 6.3.6).

    - The *alive* proposition is removed, and all states with an outgoing edge labelled by $\neg alive$ are marked as accepting.
    - Since Spot does not support state-based acceptance, a false self-loop is added to each accepting state without a successor to mark it as accepting (see Section 6.3.4).

4. **Interpret as Finite Automaton**: Finally, the Büchi automaton is interpreted as a finite automaton.

This sequence of operations was first described in [19], and later refined in [32]. Spot provides the procedure outlined above through the `spot::from_ltlf` function, which converts an $\mathtt{LTL_f}$ formula into an $\mathtt{LTL_\omega}$ formula. In this reduction, finite words are extended into infinite words, adding a new atomic proposition *alive*. Once the $\mathtt{LTL_\omega}$ formula is obtained, it can be converted into a Büchi automaton using `spot::translator` (see Section 6.3.4). Afterward, the automaton can be interpreted as a finite automaton by calling `spot::to_finite`, which produces an NFA. This last step always generates a Büchi automaton, which should be interpreted as an NFA.

6.4 ANTLR4

ANTLR4 [33] is a powerful parser generator used to define grammars for reading, processing, and translating structured text. It automatically generates parsers and tree walkers from formal grammar descriptions. In this project, ANTLR4 is employed to create a grammar for the polyhedral system and to generate its parser. This allows us to efficiently process and interpret the system's input, enabling further analysis and manipulation within the tool. ANTLR4's flexibility and widespread use in both academia and industry make it an ideal choice for this task.

6.5 Grammars, Parsers, and Lexers

When implementing a language, the goal is to create an application capable of reading input sentences and responding accordingly based on the structure of the input. A language consists of valid sentences, which are made up of phrases, and each phrase is further composed of subphrases and vocabulary symbols. Applications that execute or compute sentences are called *interpreters*. When these applications convert sentences from one language to another, they are referred to as *translators*. To process these inputs correctly, an interpreter or translator must identify valid sentences, phrases, and subphrases. This recognition process is handled by a program called a *parser* or *syntax analyser*, which ensures that the input conforms to the language's syntactical rules. Syntax refers to the set of rules that define the structure of a language. In this context, ANTLR4 provides a way to specify these rules through grammar definitions. A *grammar* consists of rules that define how phrases in a language should be structured. ANTLR translates these grammars into parsers, which are the components that process input based on the defined syntax. The grammar itself is written in ANTLR's meta-language, which is designed specifically for defining other languages. Parsing typically involves two main tasks: *lexical analysis* and *syntax analysis*. Lexical analysis, or tokenisation, involves breaking the input into smaller units known as tokens. These tokens represent the building blocks of the language, such as integers (`INT`), identifiers (`ID`), and floating-point numbers (`FLOAT`). The program responsible for this step is called the *lexer*. The lexer groups the tokens into types, which help the parser focus on the structure rather than the individual symbols. After tokenisation, the parser works with the tokens produced by the lexer to understand the overall structure of the input. The parser creates a *parse tree* or *syntax tree*, a hierarchical data structure that shows how the input matches the grammatical rules. This tree provides valuable information for further processing within the application. ANTLR generates *recursive-descent parsers* from grammar rules. These parsers consist of recursive methods for each rule in the grammar. The parsing process starts from the root (the start symbol) of the parse tree and works its way down to the leaves (the tokens). This method is known as top-down parsing, with recursive-descent parsers being one implementation of this approach.

6.6 Parse Tree Walking Mechanisms: Listeners and Visitors

When building a language application, we often need to execute specific code for each phrase or subphrase in the input. The most straightforward way to achieve this is by operating on the parse tree generated automatically by the parser. ANTLR provides two primary mechanisms for walking through the parse tree: listeners and visitors. By default, ANTLR generates a parse-tree listener interface that reacts to events triggered by the built-in tree walker. In this setup, methods in the listener act as callbacks, automatically invoked when the tree walker traverses the tree. This mechanism is simple and works well in cases where we want to handle predefined events at specific points during the tree traversal. However, there are situations where we may need more control over the traversal process. In these cases, ANTLR supports the visitor design pattern, which allows for explicit control over the walking of the parse tree. Instead of relying on automatic event callbacks, the visitor pattern enables us to visit each node of the parse tree manually by invoking methods for specific rules. To support this, ANTLR provides the `ParseTreeWalker` class, which triggers the appropriate listener methods during the tree walk. If we require more control, the `-visitor` option asks ANTLR to generate a visitor interface from the grammar, where each rule has a corresponding `visit` method. This allows us to visit the children of each node explicitly, giving us fine-grained control over how the tree is traversed. The listener and visitor mechanisms decouple grammars from application code. While the listener mechanism is useful for simple, automatic tree traversal with predefined event handling, the visitor pattern provides more flexibility and control, making it suitable for complex scenarios where custom handling of each node is required.

—7—

Polyhedral Systems: Specification Language and Representation

CONTENTS: **7.1 Polyhedral System Specification Language.** 7.1.1 Polyhedral System Grammar – 7.1.2 Parse a Polyhedral System – 7.1.3 Handling Parsing Errors. **7.2 Polyhedral System Representation.** 7.2.1 Giving Interpretations to Atomic Propositions – 7.2.2 Invariant Properties – 7.2.3 Flow Properties – 7.2.4 Computing the Pre Flow – 7.2.5 Observable Generation – 7.2.6 Extending a Polyhedral System – 7.2.7 Preserving User-Defined Variable Names in Polyhedra String Representations – 7.2.8 Constructing a Polyhedral System with `PolyhedralSystemBuilder` – 7.2.9 Class Diagram of the `system` Component.

This chapter describes how polyhedral systems are represented and specified within the tool. We begin by introducing the specification language used to define polyhedral systems, including its formal grammar and the associated parsing mechanisms. Next, we describe the internal structure of the `PolyhedralSystem` class, highlighting its key parts: atomic propositions, invariant and flow regions, and the set of observables.

7.1 Polyhedral System Specification Language

In this section, we present the grammar used to specify a polyhedral system. This is the language understood by the tool, and it is designed using ANTLR4's meta-language. This section includes the grammar, the method to parse a polyhedral system using the provided API in the `parser` component, and how the tool handles parsing and semantics errors.

7.1.1 Polyhedral System Grammar

The grammar used to specify a polyhedral system is defined as follows.

```
      LETTER ::= [a − zA − Z]
       DIGIT ::= [0 − 9]
       TIMES ::= ∗
        PLUS ::= +
       MINUS ::= −
       VARID ::= LETTER DIGIT?
UNSIGNED_INT ::= DIGIT | [1 − 9]DIGIT∗
       FLOAT ::= MINUS? UNSIGNED_INT . UNSIGNED_INT
        term ::= UNSIGNED_INT TIMES? VARID | VARID | UNSIGNED_INT
    signTerm ::= (PLUS | MINUS) term
  linearExpr ::= (PLUS | MINUS)? term (signTerm)∗
      constr ::= linearExpr (≤ | < | > | ≥ | =) linearExpr
        poly ::= { constr (& constr)∗ } | true
    powerset ::= ( poly∗ ) | ( true )
        atom ::= VARID powerset | VARID { } | VARID poly
        flow ::= FLOW poly
         inv ::= INV powerset
      system ::= atom+ flow atom* inv atom*
               | atom* flow atom+ inv atom*
               | atom* flow atom* inv atom+
               | atom+ inv atom* flow atom*
               | atom* inv atom+ flow atom*
               | atom* inv atom* flow atom+
```

Example of a Valid Polyhedral System

A valid example of a polyhedral system specification is as follows:

```
Inv ( { a >= 0 & b >= 0 } )
Flow { a + b >= -2 & a + b <= 2 & a >= -1 & a <= 1 & b >= -2 & b <= 2  & t = 1 }
p { a >= b + 1 }
q { b >= a + 1 }
t0 { t = 0 }
t1 { t <= 10 }
```

This specification defines an invariant with constraints on the variables a and b, and a flow with constraints on the variables a, b and t. It also includes atomic propositions

p and q, which specify relationships between the variables, and atomic propositions t_0 and t_1, which define the initial and time bounds. The structure of this specification adheres to the grammar defined earlier, showcasing how to represent a polyhedral system with specific constraints, flows, and atomic propositions.

7.1.2 Parse a Polyhedral System

The parsing of a polyhedral system is performed using the ANTLR4-generated parser, which processes the system specification written in the grammar defined above. This process is abstracted by the API described in Table 7.1, which provides functions to parse polyhedral system specifications from various input sources. This API is located in the `parser` component, specifically in the `system` subcomponent (`parser/system`), and is implemented in the `systemparser.h` header file. The main function for parsing a polyhedral system is `parsePolyhedralSystem`, which can accept different types of input, including strings, streams, and ANTLR input. The result of the parsing operation is encapsulated in a `PolyhedralSystemParsingResult` object, which contains either the successfully parsed system or detailed error information if the parsing fails. The `PolyhedralSystemParsingResult` object returned by these functions contains either the successfully parsed system or error information. The parsing result can be checked for success, and if successful, the parsed polyhedral system can be accessed. Otherwise, the error details can be retrieved. Below is an example of how to use the parsing functions:

Function
`parsePolyhedralSystem(std::string_view input)`
Parses the polyhedral system from a string input.
`parsePolyhedralSystem(std::istream& input)`
Parses the polyhedral system from an input stream.
`parsePolyhedralSystem(antlr4::ANTLRInputStream input)`
Parses the polyhedral system from an ANTLR input stream.

Table 7.1: Functions for parsing a polyhedral system specification.

Code 7.1 Example of parsing a polyhedral system

```
PolyhedralSystemParsingResult result =
    parsePolyhedralSystem(
        "Inv ( { a >= 0 & b >= 0 } )"
        "Flow { a + b >= -2 & a + b <= 2 & a >= -1 & a <= 1 & b >= -2 &
    b <= 2 & t = 1 }"
```

```
        "p { a >= b + 1 }"
        "q { b >= a + 1 }"
        "t0 { t = 0 }"
        "t1 { t <= 10 }"
    );

if (result)
{
    PolyhedralSystem system = *result;
    // Use the system...
}
else
{
    std::cerr << "Error while parsing Polyhedral System" << std::endl;
    std::cerr << result;
}
```

In this example, the polyhedral system specification is parsed from a string. If the parsing is successful, the resulting `PolyhedralSystem` object can be used. If parsing fails, error details are printed to the standard error stream.

7.1.3 Handling Parsing Errors

When parsing a polyhedral system specification, errors may arise if the input does not conform to the expected grammar. These errors are handled using ANTLR4's error reporting system, which provides informative messages to the user. The tool translates these errors into a standardised format to ensure consistency across different types of input. If the specification contains missing components (e.g. missing flow or invariant) or syntax errors (e.g. invalid operators or parentheses), the tool will generate an error message, indicating the location and nature of the issue. These errors are categorised into three types: lexical, syntax, and semantic errors. The tool helps identify and categorise these errors, making it easier for users to debug and resolve issues in their specifications. The `PolyhedralSystemParsingResult` class is used to encapsulate the result of parsing, as shown in Figure 7.1. If the parsing is successful, the class contains the successfully parsed polyhedral system. Otherwise, it stores a list of errors encountered during the process. The class categorises errors into lexical, syntax, and semantic errors, allowing the user to handle them appropriately. The following methods are available for working with parsing errors in the `PolyhedralSystemParsingResult` class:

- `ok()`: Checks if parsing was successful (i.e. no errors were encountered).
- `errorCount()`: Returns the total number of errors encountered during parsing.
- `lexicalErrors()`: Returns a list of lexical errors encountered during parsing.

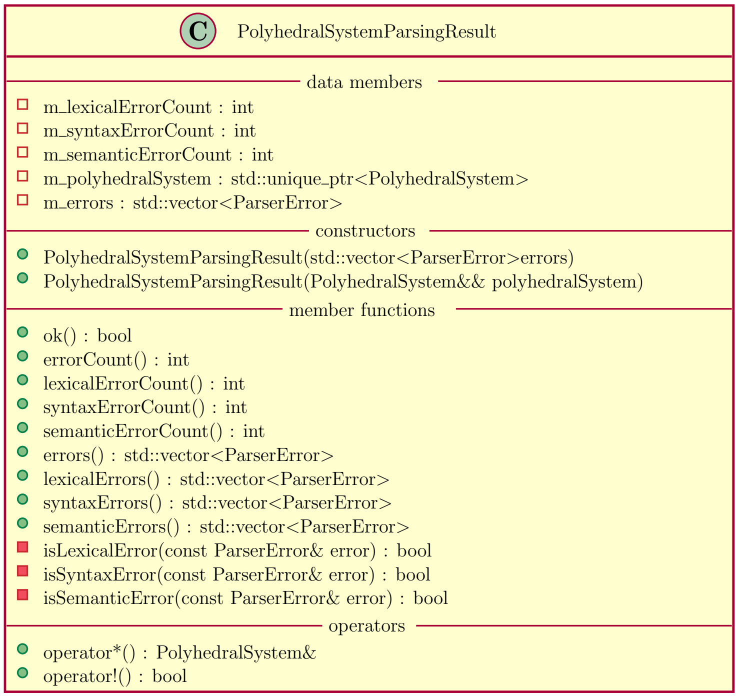
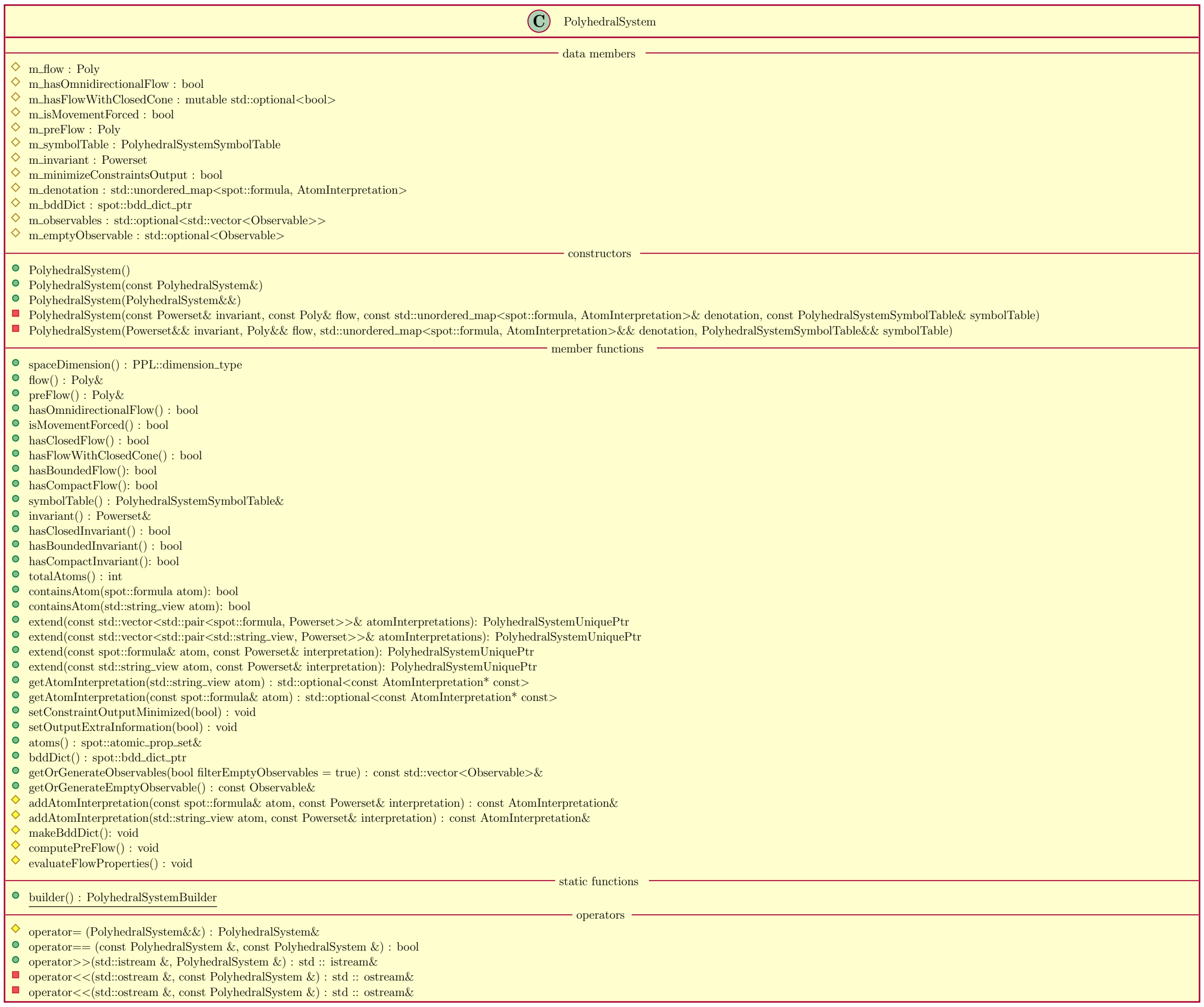

C
PolyhedralSystemParsingResult
data members
m_lexicalErrorCount : int
m_syntaxErrorCount : int
m_semanticErrorCount : int
m_polyhedralSystem : std::unique_ptr<PolyhedralSystem>
m_errors : std::vector<ParserError>
constructors
PolyhedralSystemParsingResult(std::vector<ParserError>errors)
PolyhedralSystemParsingResult(PolyhedralSystem&& polyhedralSystem)
member functions
ok() : bool
errorCount() : int
lexicalErrorCount() : int
syntaxErrorCount() : int
semanticErrorCount() : int
errors() : std::vector<ParserError>
lexicalErrors() : std::vector<ParserError>
syntaxErrors() : std::vector<ParserError>
semanticErrors() : std::vector<ParserError>
isLexicalError(const ParserError& error) : bool
isSyntaxError(const ParserError& error) : bool
isSemanticError(const ParserError& error) : bool
operators
operator*() : PolyhedralSystem&
operator!() : bool

Figure 7.1: `PolyhedralSystemParsingResult` class

- `syntaxErrors()`: Returns a list of syntax errors encountered during parsing.
- `semanticErrors()`: Returns a list of semantic errors encountered during parsing.

7.2 Polyhedral System Representation

This section illustrates how a polyhedral system, as described in Section 2, is represented. The polyhedral system is encapsulated in the `PolyhedralSystem` class, which is part of the `system` component. A visual representation of this class is shown in Figure 7.2. In the following sections, we will provide a detailed overview of this class and its relationships with other components.



```
(C) PolyhedralSystem

data members
◇ m_flow : Poly
◇ m_hasOmnidirectionalFlow : bool
◇ m_hasFlowWithClosedCone : mutable std::optional<bool>
◇ m_isMovementForced : bool
◇ m_preFlow : Poly
◇ m_symbolTable : PolyhedralSystemSymbolTable
◇ m_invariant : Powerset
◇ m_minimizeConstraintsOutput : bool
◇ m_denotation : std::unordered_map<spot::formula, AtomInterpretation>
◇ m_bddDict : spot::bdd_dict_ptr
◇ m_observables : std::optional<std::vector<Observable>>
◇ m_emptyObservable : std::optional<Observable>

constructors
● PolyhedralSystem()
● PolyhedralSystem(const PolyhedralSystem&)
● PolyhedralSystem(PolyhedralSystem&&)
■ PolyhedralSystem(const Powerset& invariant, const Poly& flow, const std::unordered_map<spot::formula, AtomInterpretation>& denotation, const PolyhedralSystemSymbolTable& symbolTable)
■ PolyhedralSystem(Powerset&& invariant, Poly&& flow, std::unordered_map<spot::formula, AtomInterpretation>&& denotation, PolyhedralSystemSymbolTable&& symbolTable)

member functions
● spaceDimension() : PPL::dimension_type
● flow() : Poly&
● preFlow() : Poly&
● hasOmnidirectionalFlow() : bool
● isMovementForced() : bool
● hasClosedFlow() : bool
● hasFlowWithClosedCone() : bool
● hasBoundedFlow(): bool
● hasCompactFlow(): bool
● symbolTable() : PolyhedralSystemSymbolTable&
● invariant() : Powerset&
● hasClosedInvariant() : bool
● hasBoundedInvariant() : bool
● hasCompactInvariant(): bool
● totalAtoms() : int
● containsAtom(spot::formula atom): bool
● containsAtom(std::string_view atom): bool
● extend(const std::vector<std::pair<spot::formula, Powerset>>& atomInterpretations): PolyhedralSystemUniquePtr
● extend(const std::vector<std::pair<std::string_view, Powerset>>& atomInterpretations): PolyhedralSystemUniquePtr
● extend(const spot::formula& atom, const Powerset& interpretation): PolyhedralSystemUniquePtr
● extend(const std::string_view atom, const Powerset& interpretation): PolyhedralSystemUniquePtr
● getAtomInterpretation(std::string_view atom) : std::optional<const AtomInterpretation* const>
● getAtomInterpretation(const spot::formula& atom) : std::optional<const AtomInterpretation* const>
● setConstraintOutputMinimized(bool) : void
● setOutputExtraInformation(bool) : void
● atoms() : spot::atomic_prop_set&
● bddDict() : spot::bdd_dict_ptr
● getOrGenerateObservables(bool filterEmptyObservables = true) : const std::vector<Observable>&
● getOrGenerateEmptyObservable() : const Observable&
◆ addAtomInterpretation(const spot::formula& atom, const Powerset& interpretation) : const AtomInterpretation&
◆ addAtomInterpretation(std::string_view atom, const Powerset& interpretation) : const AtomInterpretation&
◆ makeBddDict(): void
◆ computePreFlow() : void
◆ evaluateFlowProperties() : void

static functions
● builder() : PolyhedralSystemBuilder

operators
◆ operator= (PolyhedralSystem&&) : PolyhedralSystem&
● operator== (const PolyhedralSystem &, const PolyhedralSystem &) : bool
● operator>>(std::istream &, PolyhedralSystem &) : std :: istream&
■ operator<<(std::ostream &, const PolyhedralSystem &) : std :: ostream&
■ operator<<(std::ostream &, const PolyhedralSystem &) : std :: ostream&
```

Figure 7.2: `PolyhedralSystem` class

7.2.1 Giving Interpretations to Atomic Propositions

A key member of the `PolyhedralSystem` class is the following associative map:

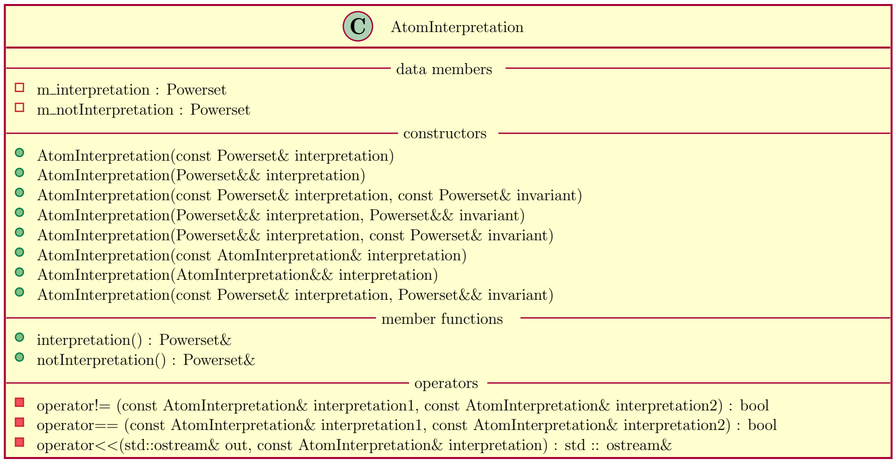
```
std::unordered_map<spot::formula, AtomInterpretation> m_denotation;
```

This map links each atomic proposition p to its interpretation $[p]$ (see Section 2.2). In Spot, an atomic proposition p is represented as a `spot::formula` (see Section 6.3.2), and its interpretation is encapsulated in the `AtomInterpretation` class shown in Figure 7.3. The `AtomInterpretation` class models the interpretation of an atomic proposition in a polyhedral system. It stores two key sets, both represented as `Powerset` objects: the set of points where the proposition holds (the interpretation), and the set of points where it does not hold (the complement). These interpretations are represented as `Powerset` objects (see Table 6.2), which are used to describe the set of points in the polyhedral system that satisfy the proposition. Additionally, the class supports intersecting the interpretation with the system's invariant region. If Inv represents the invariant region, the interpretation and its complement within the invariant are computed as follows:

$$[p]_{Inv} = [p] \cap Inv$$

$$\overline{[p]_{Inv}} = Inv \setminus [p]_{Inv}$$

The `AtomInterpretation` class provides multiple constructors, allowing the interpretation to be initialised either with or without the invariant region. It also includes methods for retrieving the interpretation and its complement. The complement of the interpretation is particularly useful for calculating observables, as demonstrated in Section 2.2.1, and more specifically in Section 7.2.5.



AtomInterpretation

☐ m_interpretation : Powerset
☐ m_notInterpretation : Powerset

● AtomInterpretation(const Powerset& interpretation)
● AtomInterpretation(Powerset&& interpretation)
● AtomInterpretation(const Powerset& interpretation, const Powerset& invariant)
● AtomInterpretation(Powerset&& interpretation, Powerset&& invariant)
● AtomInterpretation(Powerset&& interpretation, const Powerset& invariant)
● AtomInterpretation(const AtomInterpretation& interpretation)
● AtomInterpretation(AtomInterpretation&& interpretation)
● AtomInterpretation(const Powerset& interpretation, Powerset&& invariant)

● interpretation() : Powerset&
● notInterpretation() : Powerset&

■ operator!= (const AtomInterpretation& interpretation1, const AtomInterpretation& interpretation2) : bool
■ operator== (const AtomInterpretation& interpretation1, const AtomInterpretation& interpretation2) : bool
■ operator<<(std::ostream& out, const AtomInterpretation& interpretation) : std :: ostream&

Figure 7.3: `AtomInterpretation` class

In addition, Table 7.2 provides a summary of the methods in the `PolyhedralSystem` class for accessing atomic propositions and their interpretations. These methods allow for easy retrieval of the atomic propositions, checking if a specific atom exists, and getting their corresponding interpretations. The table below summarises these methods:

Method	**Description**
`atoms()`	Returns the set of atomic propositions in the system.
`containsAtom(spot::formula atom)`	Checks if a specific atomic proposition exists. Accepts a `spot::formula` for the proposition.
`containsAtom(std::string_view atom)`	Checks if an atomic proposition by name exists. Accepts the name as a string.
`totalAtoms()`	Returns the total number of atomic propositions in the system.
`getAtomInterpretation(std::string_view atom)`	Retrieves the interpretation by name. Returns `std::nullopt` if the atom doesn't exist.
`getAtomInterpretation(spot::formula& atom)`	Retrieves the interpretation of an atomic proposition. Returns `std::nullopt` if the proposition is not found.

Table 7.2: Methods for accessing atomic propositions and their interpretations in the `PolyhedralSystem` class.

7.2.2 Invariant Properties

In the `PolyhedralSystem` class, the invariant region of the system is represented as a `Powerset`. The invariant region can be used to filter or constrain the possible states or trajectories of the system. The `PolyhedralSystem` class provides several methods for accessing and analysing the invariant. These methods allow users to query the properties of the invariant region, such as whether it is bounded, closed, or compact. The following table summarises the methods in the `PolyhedralSystem` class related to the invariant region:

Method	**Description**
`invariant()`	Returns the invariant region of the polyhedral system as a `Powerset`.
`hasBoundedInvariant()`	Checks if the invariant region is bounded. Returns `true` if bounded, `false` otherwise.
`hasClosedInvariant()`	Checks if the invariant region is topologically closed. Returns `true` if closed, `false` otherwise.
`hasCompactInvariant()`	Checks if the invariant region is compact. Returns `true` if compact, `false` otherwise.

Table 7.3: Methods for accessing and analyzing the invariant region in the `PolyhedralSystem` class.

7.2.3 Flow Properties

The `PolyhedralSystem` class also provides methods for accessing and analysing the flow properties of the system. The flow constraint defines the region in which the system can move, and the pre-flow constraint is its reflection. Several methods allow you to query the properties of the flow, such as whether the flow is omnidirectional, bounded, closed, or compact. Additionally, some methods provide insights into the geometric properties of the flow, such as whether the cone of the flow is closed or whether movement is forced in the system. The following table summarises the methods related to flow properties in the `PolyhedralSystem` class:

Method	**Description**
`flow()`	Returns the flow constraint of the polyhedral system as a convex polyhedron.
`preFlow()`	Returns the pre-flow polyhedron, which is the reflection of the flow constraint.
`hasOmnidirectionalFlow()`	Checks if the flow constraint is omnidirectional. Returns `true` if omnidirectional, `false` otherwise.
`hasBoundedFlow()`	Checks if the flow constraint is bounded. Returns `true` if bounded, `false` otherwise.
`hasClosedFlow()`	Checks if the flow constraint is topologically closed. Returns `true` if closed, `false` otherwise.
`hasCompactFlow()`	Checks if the flow constraint is compact. Returns `true` if compact, `false` otherwise.
`hasFlowWithClosedCone()`	Checks if the cone of the flow is closed. Returns `true` if closed, `false` otherwise.
`isMovementForced()`	Checks if movement is forced in the system (i.e., the origin is not in the flow closure). Returns `true` if forced, `false` otherwise.

Table 7.4: Methods for accessing and analysing the flow properties in the `PolyhedralSystem` class.

7.2.4 Computing the Pre Flow

In the construction of a polyhedral system, the pre-flow is computed and stored. The pre-flow is used to calculate the positive pre-flow of a polyhedron P (see Section 2.5.3). Computing the pre-flow is a straightforward task. It is achieved by applying a reflection transformation to the convex polyhedron $Flow$. The reflection is performed by negating each coordinate of the polyhedron. Specifically, for each dimension $i \in \{0, \ldots, n-1\}$, the coordinate x_i is replaced with $-x_i$. Below is the code for the function that applies the reflection transformation to a polyhedron.

Code 7.2 Applying Reflection Transformation to a Polyhedron

```
inline Poly& reflectionAffineImage(Poly& polyhedron) {
    const PPL::dimension_type spaceDimension { polyhedron.space_dimension() };

    for (PPL::dimension_type dim {}; dim < spaceDimension; ++dim) {
        const PPL::Variable variable { dim };
        polyhedron.affine_image(variable, -variable);
    }

    return polyhedron;
}
```

The transformation is done in place, meaning that the original polyhedron is directly modified and no new polyhedron is created. This function can be found in the `geometry` component, specifically in the `reflection.h` module.

7.2.5 Observable Generation

In the `PolyhedralSystem`, observables represent sets of atomic propositions, and their interpretations are computed based on the atomic propositions defined in the system. These observables are useful for identifying specific regions of the state space where particular properties hold. The process of generating observables can be computationally expensive, as the number of possible observables grows exponentially with the number of atomic propositions. Specifically, if n is the number of atomic propositions, the total number of observables is 2^n. Due to this exponential growth, observables are generated lazily (on-demand). A visual representation of the `Observable` class is shown in Figure 7.4. This class encapsulates the set of atomic propositions that form an observable and the corresponding polyhedron that defines its interpretation. The Table 7.5 describe the two key methods used for generating observables in the `PolyhedralSystem` class:

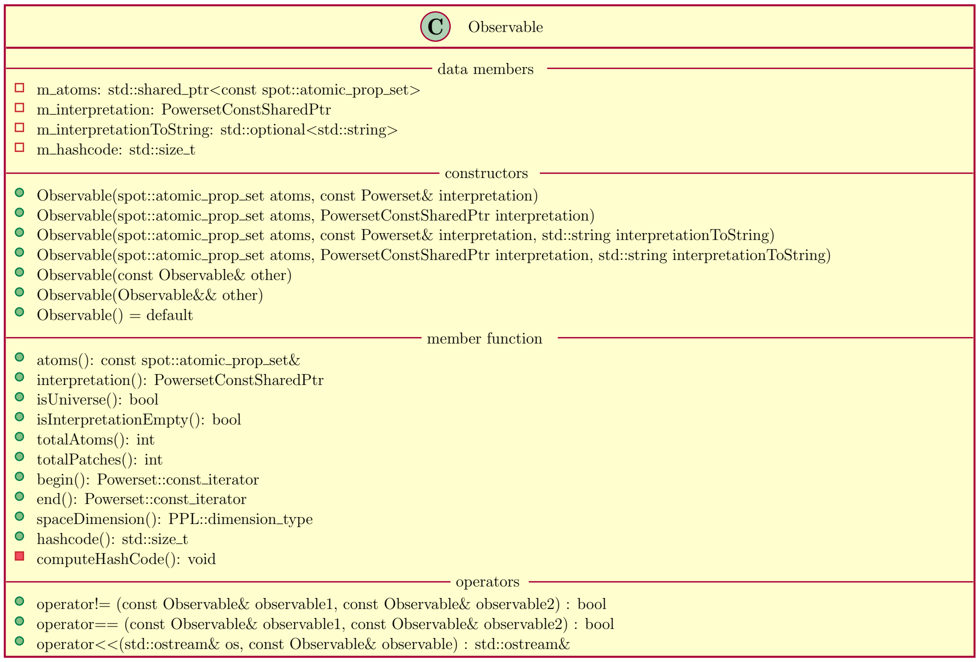

C
Observable
data members
m_atoms: std::shared_ptr<const spot::atomic_prop_set>
m_interpretation: PowersetConstSharedPtr
m_interpretationToString: std::optional<std::string>
m_hashcode: std::size_t
constructors
Observable(spot::atomic_prop_set atoms, const Powerset& interpretation)
Observable(spot::atomic_prop_set atoms, PowersetConstSharedPtr interpretation)
Observable(spot::atomic_prop_set atoms, const Powerset& interpretation, std::string interpretationToString)
Observable(spot::atomic_prop_set atoms, PowersetConstSharedPtr interpretation, std::string interpretationToString)
Observable(const Observable& other)
Observable(Observable&& other)
Observable() = default
member function
atoms(): const spot::atomic_prop_set&
interpretation(): PowersetConstSharedPtr
isUniverse(): bool
isInterpretationEmpty(): bool
totalAtoms(): int
totalPatches(): int
begin(): Powerset::const_iterator
end(): Powerset::const_iterator
spaceDimension(): PPL::dimension_type
hashcode(): std::size_t
computeHashCode(): void
operators
operator!= (const Observable& observable1, const Observable& observable2) : bool
operator== (const Observable& observable1, const Observable& observable2) : bool
operator<<(std::ostream& os, const Observable& observable) : std::ostream&

Figure 7.4: `Observable` class

Method	**Description**
`getOrGenerateObservables()`	Computes all possible subsets of atomic propositions and their interpretations. Uses bitwise operations to iterate through subsets and compute interpretations.
`getOrGenerateEmptyObservable()`	Computes the empty observable where none of the atomic propositions hold. Intersects the negations of all atomic propositions to obtain the empty observable.

Table 7.5: Methods for generating observables in the `PolyhedralSystem`.

As generating all observables at once can be computationally expensive, the process is done lazily. Observables are generated only when needed. Each observable is computed and stored only once, and later calls to retrieve it return the precomputed result.

7.2.6 Extending a Polyhedral System

The `PolyhedralSystem` class provides several methods for extending the system by adding new atomic propositions and their corresponding interpretations. These methods allow you to build a new `PolyhedralSystem` by adding atomic propositions along with their interpretations (represented as `Powerset`s). These methods can be used to extend an existing system either by providing a vector of atomic proposition-interpretation pairs or by passing individual atomic propositions with their interpretations. This functionality is useful when new atomic propositions need to be introduced into an already existing polyhedral system (like ***brink***, see Section 4.6.1). The extension process involves creating a new system while keeping the original polyhedral structure and adding the new atomic propositions and their respective interpretations. The following methods are used to extend a `PolyhedralSystem` with new atomic propositions and interpretations:

- `extend(const std::vector<std::pair<spot::formula, Powerset>>& atomInterpretations)`: Creates a new system by extending the original with a vector of atomic propositions and their interpretations.

- `extend(const std::vector<std::pair<std::string_view, Powerset>>& atomInterpretations)`: Same as above, but accepts atomic propositions as strings.

- `extend(const spot::formula& atom, const Powerset& interpretation)`: Extends the system with a single atomic proposition and its interpretation.

- `extend(std::string_view atom, const Powerset& interpretation)`: Extends the system with a single atomic proposition (given as a string) and its interpretation.

7.2.7 Preserving User-Defined Variable Names in Polyhedra String Representations

As discussed in Section 7.1, users of the tool can represent a `PolyhedralSystem` through a specific language. Each atomic proposition is assigned a `Poly` or `Powerset`, and users can assign custom identifiers to the variables. However, PPL (Parma Polyhedra Library) assigns each variable an internally generated identifier. To output results with the identifiers chosen by the user, it is necessary to store the mapping between the user-defined identifiers and the variables represented by PPL. This mapping is maintained in the `PolyhedralSystemSymbolTable` class, which also stores the atomic propositions chosen by the user. `PolyhedralSystemSymbolTable` ensures that the user-defined variable names are correctly associated with the corresponding dimensions in the state space, and it helps in generating human-readable output by providing the correct variable names when representing polyhedra.

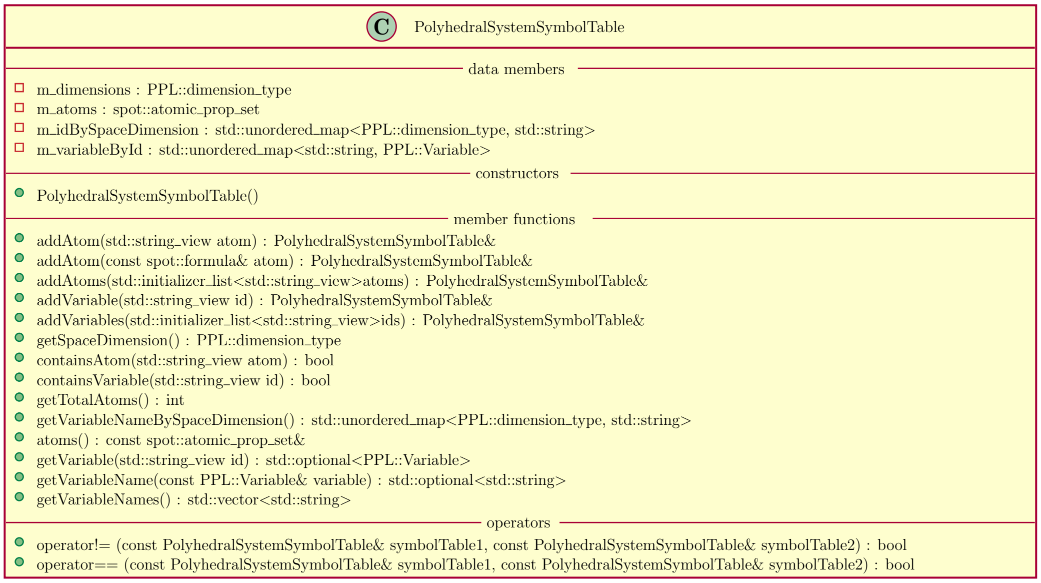

C
PolyhedralSystemSymbolTable
data members
m_dimensions : PPL::dimension_type
m_atoms : spot::atomic_prop_set
m_idBySpaceDimension : std::unordered_map<PPL::dimension_type, std::string>
m_variableById : std::unordered_map<std::string, PPL::Variable>
constructors
PolyhedralSystemSymbolTable()
member functions
addAtom(std::string_view atom) : PolyhedralSystemSymbolTable&
addAtom(const spot::formula& atom) : PolyhedralSystemSymbolTable&
addAtoms(std::initializer_list<std::string_view>atoms) : PolyhedralSystemSymbolTable&
addVariable(std::string_view id) : PolyhedralSystemSymbolTable&
addVariables(std::initializer_list<std::string_view>ids) : PolyhedralSystemSymbolTable&
getSpaceDimension() : PPL::dimension_type
containsAtom(std::string_view atom) : bool
containsVariable(std::string_view id) : bool
getTotalAtoms() : int
getVariableNameBySpaceDimension() : std::unordered_map<PPL::dimension_type, std::string>
atoms() : const spot::atomic_prop_set&
getVariable(std::string_view id) : std::optional<PPL::Variable>
getVariableName(const PPL::Variable& variable) : std::optional<std::string>
getVariableNames() : std::vector<std::string>
operators
operator!= (const PolyhedralSystemSymbolTable& symbolTable1, const PolyhedralSystemSymbolTable& symbolTable2) : bool
operator== (const PolyhedralSystemSymbolTable& symbolTable1, const PolyhedralSystemSymbolTable& symbolTable2) : bool

Figure 7.5: `PolyhedralSystemSymbolTable` class

The table is constructed during the parsing process. The functions responsible for converting PPL objects into string representations are part of the `PPLOutput` namespace. They ensure that the variable names used in the output match the user-defined names, providing a more intuitive and readable representation of polyhedra and constraints. Below are some of the main functions:

- `toString(const Powerset&, const PolyhedralSystemSymbolTable& symbolTable, bool minimizeConstraints)`: Converts a `Powerset` into a human-readable string. The `minimizeConstraints` flag can be set to `true` to simplify constraints before conversion.

- `toString(const Poly&, const PolyhedralSystemSymbolTable& symbolTable, bool minimizeConstraints)`: Converts a `Poly` into a human-readable string. The `minimizeConstraints` flag can be used to reduce redundancy in the printed constraints.

- `toString(const PPL::Constraint&, const std::unordered_map<PPL::dimension_type, std::string>& symbolTable)`: Converts a single `Constraint` into a string using the provided symbol table for variable names.

- `toString(const PPL::Variable&, const std::unordered_map<PPL::dimension_type, std::string>& symbolTable)`: Converts a `Variable` into a string using the symbol table.

The formatting behaviour of these functions can also be influenced by a few global flags. The global flag `g_andConstraints` determines the separator used between multiple constraints, and its default value is `"&"`. The flag `g_geqSign` specifies the symbol used to represent inequalities; by default, it is set to `">="`. Finally, `g_wrapConstraintInRoundBrackets` controls whether each constraint should be enclosed in round brackets. Its default value is `false`.

Adding New Variables to the Symbol Table

`PolyhedralSystemSymbolTable` allows the addition of new variables to the symbol table. Each new variable added increases the space dimension by 1. The symbol table stores a bidirectional mapping between user-defined variable names and their corresponding PPL variables. Given the user-defined variable name (as a `std::string`), it is possible to retrieve the corresponding PPL variable. This feature ensures that users can interact with the system using their own naming conventions while maintaining consistency with the underlying PPL implementation. The Table 7.6 shows a summary of the key methods in the `PolyhedralSystemSymbolTable` class.

Method	**Description**
`addVariable`	Adds a new variable with a user-defined name to the symbol table.
`addVariables`	Adds multiple variables with user-defined names to the symbol table.
`addAtom`	Adds an atomic proposition to the symbol table.
`containsVariable`	Checks if a variable with the given name exists in the symbol table.
`containsAtom`	Checks if an atomic proposition with the given name exists in the symbol table.
`getVariable`	Retrieves the corresponding PPL variable for a user-defined name.
`getVariableName`	Retrieves the user-defined name corresponding to a PPL variable.
`getVariableNames`	Returns a list of all user-defined variable names in the symbol table.
`getTotalAtoms`	Returns the total number of atomic propositions in the symbol table.
`spaceDimension`	Returns the total number of dimensions in the state space.

Table 7.6: Methods for interacting with the `PolyhedralSystemSymbolTable`.

7.2.8 Constructing a Polyhedral System with `PolyhedralSystemBuilder`

The `PolyhedralSystemBuilder` class provides an elegant and structured way to construct a `PolyhedralSystem` object. It ensures that all necessary components such as flow, invariant, denotation, and symbol table are properly set and validated before constructing the final polyhedral system. The builder pattern allows for a step-by-step construction process, ensuring that all parts of the system are configured properly. This approach also makes the code more readable and less error-prone by guiding the user through the construction process in a logical sequence. One of the advantages of using the builder pattern is that it ensures all dependencies between components (e.g. the flow and the invariant) are respected, preventing mistakes that could occur when constructing the system manually. The following example demonstrates how to construct a polyhedral system using the `PolyhedralSystemBuilder` class, showcasing how to set up variables, atomic propositions, and constraints in a clean and easy-to-read manner.

Code 7.3 Example of Constructing a Polyhedral System with `PolyhedralSystemBuilder`

```cpp
using namespace PPLUtils;

PolyhedralSystemSymbolTable symbolTable {};
symbolTable
    .addVariables({ "X", "Y" })
    .addAtoms({ "p", "q" });

PPL::Variable x { *symbolTable.getVariable("x") };
PPL::Variable y { *symbolTable.getVariable("y") };

PolyhedralSystem expectedPolyhedralSystem {
    PolyhedralSystem::builder()
        .flow(poly({ x + 0*y <= 4 }))
        .denotation({
            { ap("p"), powerset({{ x < 3, y <= 3 }, { x < 3, y <= 10 }}) },
            { ap("q"), powerset({{ x > 3, y >= 4 }})}
        })
        .invariant(powerset({{ x + y >= 3, y >= 4 }}))
        .symbolTable(symbolTable)
        .build()
};
```

In this example:

- The `PolyhedralSystemSymbolTable` is used to manage user-defined variable names and atomic propositions.

- The `PolyhedralSystemBuilder` class is used to build the polyhedral system step by step, using the variables and atomic propositions defined earlier.
- Finally, the `build()` method constructs the polyhedral system using the provided components.

This approach allows for more readable and error-free code while maintaining flexibility. The `PolyhedralSystemBuilder` ensures that all required components are set correctly before constructing the final polyhedral system, while also allowing easy modifications and extensions.

7.2.9 Class Diagram of the `system` Component

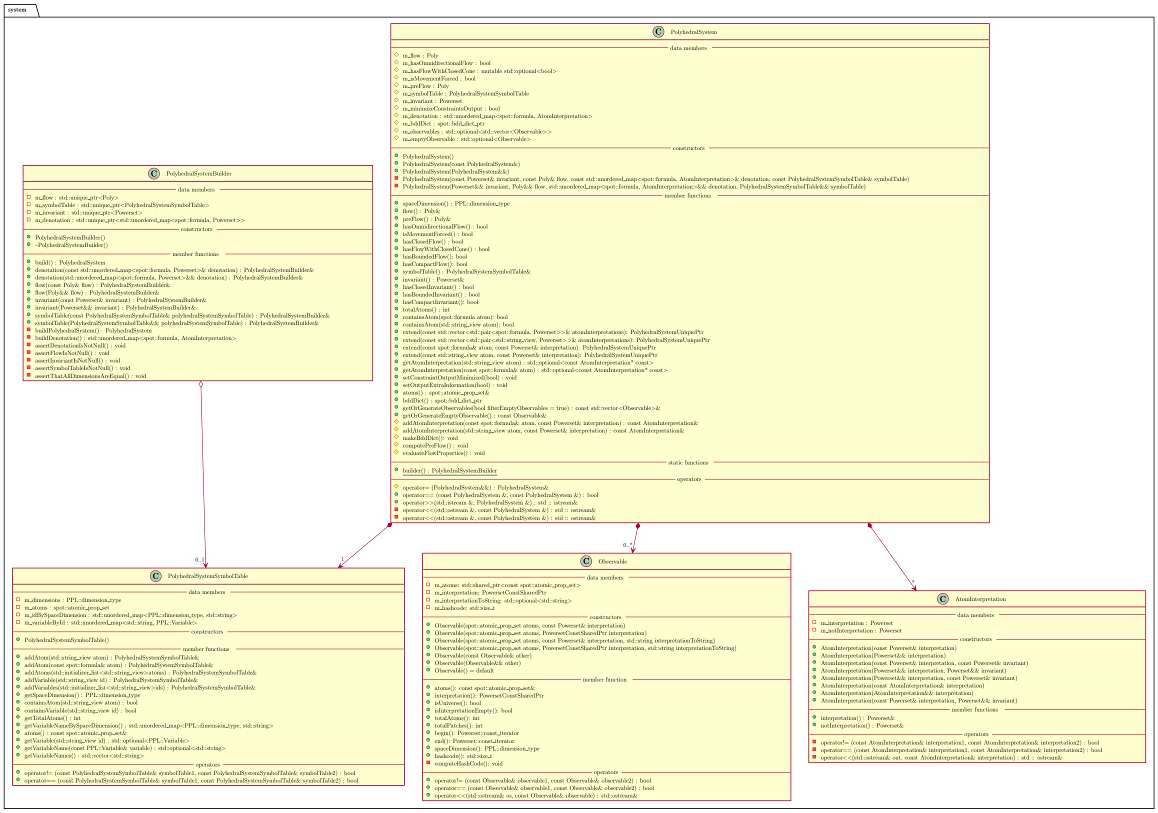

system
PolyhedralSystem
data members
m_flow : Poly
m_hasOmnidirectionalFlow : bool
m_hasFlowWithClosedCone : mutable std::optional<bool>
m_isMovementForced : bool
m_preFlow : Poly
m_symbolTable : PolyhedralSystemSymbolTable
m_invariant : Powerset
m_minimizeConstraintsOutput : bool
m_denotation : std::unordered_map<spot::formula, AtomInterpretation>
m_bddDict : spot::bdd_dict_ptr
m_observables : std::optional<std::vector<Observable>>
m_emptyObservable : std::optional<Observable>
constructors
PolyhedralSystem()
PolyhedralSystem(const PolyhedralSystem&)
PolyhedralSystem(PolyhedralSystem&&)
PolyhedralSystem(const Powerset& invariant, const Poly& flow, const std::unordered_map<spot::formula, AtomInterpretation>& denotation, const PolyhedralSystemSymbolTable& symbolTable)
PolyhedralSystem(Powerset&& invariant, Poly&& flow, std::unordered_map<spot::formula, AtomInterpretation>&& denotation, PolyhedralSystemSymbolTable&& symbolTable)
member functions
spaceDimension() : PPL::dimension_type
flow() : Poly&
preFlow() : Poly&
hasOmnidirectionalFlow() : bool
isMovementForced() : bool
hasClosedFlow() : bool
hasFlowWithClosedCone() : bool
hasBoundedFlow(): bool
hasCompactFlow(): bool
symbolTable() : PolyhedralSystemSymbolTable&
invariant() : Powerset&
hasClosedInvariant() : bool
hasBoundedInvariant() : bool
hasCompactInvariant(): bool
totalAtoms() : int
containsAtom(spot::formula atom): bool
containsAtom(std::string_view atom): bool
extend(const std::vector<std::pair<spot::formula, Powerset>>& atomInterpretations): PolyhedralSystemUniquePtr
extend(const std::vector<std::pair<std::string_view, Powerset>>& atomInterpretations): PolyhedralSystemUniquePtr
extend(const spot::formula& atom, const Powerset& interpretation): PolyhedralSystemUniquePtr
extend(const std::string_view atom, const Powerset& interpretation): PolyhedralSystemUniquePtr
getAtomInterpretation(std::string_view atom) : std::optional<const AtomInterpretation* const>
getAtomInterpretation(const spot::formula& atom) : std::optional<const AtomInterpretation* const>
setConstraintOutputMinimized(bool) : void
setOutputExtraInformation(bool) : void
atoms() : spot::atomic_prop_set&
bddDict() : spot::bdd_dict_ptr
getOrGenerateObservables(bool filterEmptyObservables = true) : const std::vector<Observable>&
getOrGenerateEmptyObservable() : const Observable&
addAtomInterpretation(const spot::formula& atom, const Powerset& interpretation) : const AtomInterpretation&
addAtomInterpretation(std::string_view atom, const Powerset& interpretation) : const AtomInterpretation&
makeBddDict(): void
computePreFlow() : void
evaluateFlowProperties() : void
static functions
builder() : PolyhedralSystemBuilder
operators
operator= (PolyhedralSystem&&) : PolyhedralSystem&
operator== (const PolyhedralSystem &, const PolyhedralSystem &) : bool
operator>>(std::istream &, PolyhedralSystem &) : std :: istream&
operator<<(std::ostream &, const PolyhedralSystem &) : std :: ostream&
operator<<(std::ostream &, const PolyhedralSystem &) : std :: ostream&
PolyhedralSystemBuilder
data members
m_flow : std::unique_ptr<Poly>
m_symbolTable : std::unique_ptr<PolyhedralSystemSymbolTable>
m_invariant : std::unique_ptr<Powerset>
m_denotation : std::unique_ptr<std::unordered_map<spot::formula, Powerset>>
constructors
PolyhedralSystemBuilder()
~PolyhedralSystemBuilder()
member functions
build() : PolyhedralSystem
denotation(const std::unordered_map<spot::formula, Powerset>& denotation) : PolyhedralSystemBuilder&
denotation(std::unordered_map<spot::formula, Powerset>&& denotation) : PolyhedralSystemBuilder&
flow(const Poly& flow) : PolyhedralSystemBuilder&
flow(Poly&& flow) : PolyhedralSystemBuilder&
invariant(const Powerset& invariant) : PolyhedralSystemBuilder&
invariant(Powerset&& invariant) : PolyhedralSystemBuilder&
symbolTable(const PolyhedralSystemSymbolTable& polyhedralSystemSymbolTable) : PolyhedralSystemBuilder&
symbolTable(PolyhedralSystemSymbolTable&& polyhedralSystemSymbolTable) : PolyhedralSystemBuilder&
buildPolyhedralSystem() : PolyhedralSystem
buildDenotation() : std::unordered_map<spot::formula, AtomInterpretation>
assertDenotationIsNotNull() : void
assertFlowIsNotNull() : void
assertInvariantIsNotNull() : void
assertSymbolTableIsNotNull() : void
assertThatAllDimensionsAreEqual() : void
0..1
1
0..*
*
PolyhedralSystemSymbolTable
data members
m_dimensions : PPL::dimension_type
m_atoms : spot::atomic_prop_set
m_idBySpaceDimension : std::unordered_map<PPL::dimension_type, std::string>
m_variableById : std::unordered_map<std::string, PPL::Variable>
constructors
PolyhedralSystemSymbolTable()
member functions
addAtom(std::string_view atom) : PolyhedralSystemSymbolTable&
addAtom(const spot::formula& atom) : PolyhedralSystemSymbolTable&
addAtoms(std::initializer_list<std::string_view>atoms) : PolyhedralSystemSymbolTable&
addVariable(std::string_view id) : PolyhedralSystemSymbolTable&
addVariables(std::initializer_list<std::string_view>ids) : PolyhedralSystemSymbolTable&
getSpaceDimension() : PPL::dimension_type
containsAtom(std::string_view atom) : bool
containsVariable(std::string_view id) : bool
getTotalAtoms() : int
getVariableNameBySpaceDimension() : std::unordered_map<PPL::dimension_type, std::string>
atoms() : const spot::atomic_prop_set&
getVariable(std::string_view id) : std::optional<PPL::Variable>
getVariableName(const PPL::Variable& variable) : std::optional<std::string>
getVariableNames() : std::vector<std::string>
operators
operator!= (const PolyhedralSystemSymbolTable& symbolTable1, const PolyhedralSystemSymbolTable& symbolTable2) : bool
operator== (const PolyhedralSystemSymbolTable& symbolTable1, const PolyhedralSystemSymbolTable& symbolTable2) : bool
Observable
data members
m_atoms: std::shared_ptr<const spot::atomic_prop_set>
m_interpretation: PowersetConstSharedPtr
m_interpretationToString: std::optional<std::string>
m_hashcode: std::size_t
constructors
Observable(spot::atomic_prop_set atoms, const Powerset& interpretation)
Observable(spot::atomic_prop_set atoms, PowersetConstSharedPtr interpretation)
Observable(spot::atomic_prop_set atoms, const Powerset& interpretation, std::string interpretationToString)
Observable(spot::atomic_prop_set atoms, PowersetConstSharedPtr interpretation, std::string interpretationToString)
Observable(const Observable& other)
Observable(Observable&& other)
Observable() = default
member function
atoms(): const spot::atomic_prop_set&
interpretation(): PowersetConstSharedPtr
isUniverse(): bool
isInterpretationEmpty(): bool
totalAtoms(): int
totalPatches(): int
begin(): Powerset::const_iterator
end(): Powerset::const_iterator
spaceDimension(): PPL::dimension_type
hashcode(): std::size_t
computeHashCode(): void
operators
operator!= (const Observable& observable1, const Observable& observable2) : bool
operator== (const Observable& observable1, const Observable& observable2) : bool
operator<<(std::ostream& os, const Observable& observable) : std::ostream&
AtomInterpretation
data members
m_interpretation : Powerset
m_notInterpretation : Powerset
constructors
AtomInterpretation(const Powerset& interpretation)
AtomInterpretation(Powerset&& interpretation)
AtomInterpretation(const Powerset& interpretation, const Powerset& invariant)
AtomInterpretation(Powerset&& interpretation, Powerset&& invariant)
AtomInterpretation(Powerset&& interpretation, const Powerset& invariant)
AtomInterpretation(const AtomInterpretation& interpretation)
AtomInterpretation(AtomInterpretation&& interpretation)
AtomInterpretation(const Powerset& interpretation, Powerset&& invariant)
member functions
interpretation() : Powerset&
notInterpretation() : Powerset&
operators
operator!= (const AtomInterpretation& interpretation1, const AtomInterpretation& interpretation2) : bool
operator== (const AtomInterpretation& interpretation1, const AtomInterpretation& interpretation2) : bool
operator<<(std::ostream& out, const AtomInterpretation& interpretation) : std :: ostream&

Figure 7.6: `system` component class diagram

–8–

Representing and Discretising `RTL` Formulae

CONTENTS: **8.1 `RTL` Specification Language.** 8.1.1 `RTL` Grammar – 8.1.2 Parse an `RTL` Formula – 8.1.3 `RTL` Parsing Result Representation. **8.2 `RTL` Formula Utilities and Constants: the `formula` Component.** 8.2.1 Utilities and Operations on Formulae – 8.2.2 `RTL` Predefined Constants and Special Formulae – 8.2.3 Discretisation Transformation Rules – 8.2.4 Optimising Discretisation Process for Finite-Time Semantics.

This chapter describes how `RTL` formulae are represented, parsed, and discretised to enable model checking over polyhedral systems. To integrate `RTL` formulae into the verification pipeline, they must be formally defined, syntactically validated, and discretised into equivalent `LTL` formulae. This process requires a well-defined grammar, a robust parser, and a set of discretisation rules that preserve semantics. We begin by introducing the syntax and grammar of `RTL` formulae.

8.1 `RTL` Specification Language

The `RTL` specification language is designed to formally represent real-time temporal properties over polyhedral systems. This section provides an in-depth look at the grammar, syntax, and parsing mechanisms for `RTL` formulae.

8.1.1 RTL Grammar

$$
\begin{aligned}
\texttt{CONSTANT} &::= 0 \mid 1 \\
\texttt{LETTER} &::= [a-zA-Z] \\
\texttt{DIGIT} &::= [0-9] \\
\texttt{ATOM} &::= \texttt{LETTER}\ \texttt{DIGIT?}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{bformula} &::= \texttt{CONSTANT} \\
&\mid \texttt{ATOM} \\
&\mid \texttt{ATOM} = 0 \\
&\mid \texttt{ATOM} = 1 \\
&\mid (\texttt{bformula}) \\
&\mid \texttt{bformula}\ \mathrm{xor}\ \texttt{bformula} \\
&\mid \texttt{bformula}\ \&\ \texttt{bformula} \\
&\mid !\ \texttt{bformula} \\
&\mid \texttt{bformula} \leftrightarrow \texttt{bformula} \\
&\mid \texttt{bformula} \longrightarrow \texttt{bformula} \\
&\mid \texttt{bformula} \mid \texttt{bformula}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{tformula} &::= \texttt{bformula} \\
&\mid \mathsf{X}\ \texttt{tformula} \\
&\mid \mathsf{X}[!]\ \texttt{tformula} \\
&\mid (\texttt{tformula}) \\
&\mid !\ \texttt{tformula} \\
&\mid \texttt{tformula}\ \&\ \texttt{tformula} \\
&\mid \texttt{tformula} \mid \texttt{tformula} \\
&\mid \texttt{tformula} \longrightarrow \texttt{tformula} \\
&\mid \texttt{tformula}\ \mathrm{xor}\ \texttt{tformula} \\
&\mid \texttt{tformula} \leftrightarrow \texttt{tformula} \\
&\mid \texttt{tformula}\ \mathsf{U}\ \texttt{tformula} \\
&\mid \texttt{tformula}\ \mathsf{W}\ \texttt{tformula} \\
&\mid \texttt{tformula}\ \mathsf{R}\ \texttt{tformula} \\
&\mid \texttt{tformula}\ \mathsf{M}\ \texttt{tformula} \\
&\mid \mathsf{X}\ \texttt{tformula} \\
&\mid \mathsf{X}[!]\ \texttt{tformula} \\
&\mid \mathsf{F}\ \texttt{tformula} \\
&\mid \mathsf{G}\ \texttt{tformula}
\end{aligned}
$$

8.1.2 Parse an RTL Formula

The parsing of an `RTL` formula is handled by the `parsertl.h` module, which is part of the `parser/rtl` component. The module offers several functions for parsing `RTL` formulae from different input sources. The parsing process follows the grammar and ensures that the formula conforms to the expected structure. The module provides the following functions for parsing `RTL` formulae:

The `parseRtl` function parses an `RTL` formula from a string input and returns a `RtlParsingResult` object, which contains either the parsed formula or any errors encountered during parsing. The second version of the function allows validation of atomic propositions by accepting an `allowedAtomicPropositions` parameter, ensuring that the formula uses only atomic propositions defined in the polyhedral system. The third version of the function accepts an input stream rather than a string, providing more flexibility for parsing `RTL` formulae from various sources. The Code 8.1 is an example of how to use the parsing functions.

Function
`parseRtl(std::string_view rtl)` Parses the `RTL` formula from a string input.
`parseRtl(std::string_view rtl, allowedAtomicPropositions)` Parses the `RTL` formula with atomic proposition validation.
`parseRtl(std::istream& rtl, allowedAtomicPropositions)` Parses the `RTL` formula from an input stream with atomic proposition validation.

Table 8.1: Functions for parsing RTL formulae in the `parsertl.h` module.

Code 8.1 Example of parsing an RTL formula

```
std::string rtl { "G(F(p0)) & F(!p1) & F(G(F(G(p2 U !p3))))" };
spot::atomic_prop_set allowedAtomicProp { AP({ "p0", "p1", "p2" }) };
RtlParsingResult rtlParsingResult { parseRtl(rtl, allowedAtomicProp) };

if (rtlParsingResult)
{
    // Parsing succeeded, access the parsed formula
    auto parsedFormula = *rtlParsingResult;
    // Use parsedFormula...
}
else
{
    std::cerr << "Error while parsing RTL Formula" << std::endl;
    std::cerr << rtlParsingResult; // Output the errors
}
```

8.1.3 `RTL` Parsing Result Representation

The result of parsing an RTL formula is represented by the `RtlParsingResult` class. This class encapsulates the outcome of parsing, providing access to the successfully parsed formula (if parsing is successful), any parser errors encountered, and information about the validity of the formula and its atomic propositions. The `RtlParsingResult` class categorises parsing issues into different types:

- *Parser Errors*: Errors that occurred during the parsing process, such as syntax mistakes.
- *Illegal Atomic Propositions*: A list of atomic propositions used in the formula that are not defined in the polyhedral system.

- *LTL Validation*: Whether the formula is a valid Linear Temporal Logic (LTL) formula.

The result can be checked for success using the `ok()` method. If parsing was successful, the formula can be accessed via the dereference operator `*`. If errors were encountered, they can be retrieved through accessor methods like `parserErrors()`, `illegalAtoms()`, or `isLtl()`. The `RtlParsingResult` class provides several methods to inspect and interact with the parsing result. If parsing fails, detailed information about errors and illegal atoms is available. `RtlParsingResult` also supports streaming error reporting through the overloaded output stream operator, which prints all relevant information, including parsing errors and invalid atomic propositions.

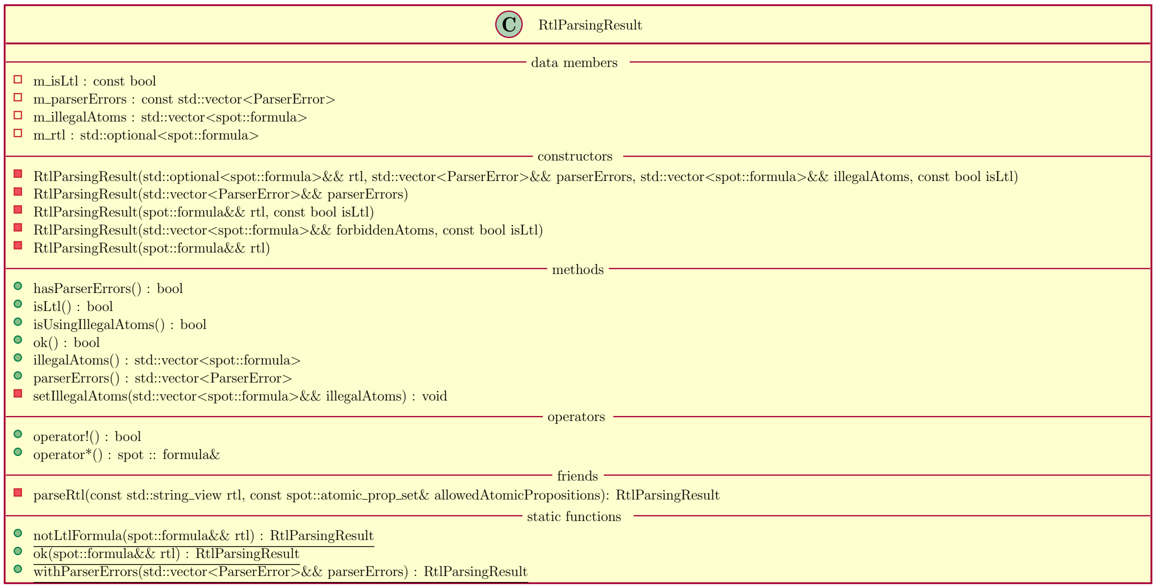

C
RtlParsingResult
data members
m_isLtl : const bool
m_parserErrors : const std::vector<ParserError>
m_illegalAtoms : std::vector<spot::formula>
m_rtl : std::optional<spot::formula>
constructors
RtlParsingResult(std::optional<spot::formula>&& rtl, std::vector<ParserError>&& parserErrors, std::vector<spot::formula>&& illegalAtoms, const bool isLtl)
RtlParsingResult(std::vector<ParserError>&& parserErrors)
RtlParsingResult(spot::formula&& rtl, const bool isLtl)
RtlParsingResult(std::vector<spot::formula>&& forbiddenAtoms, const bool isLtl)
RtlParsingResult(spot::formula&& rtl)
methods
hasParserErrors() : bool
isLtl() : bool
isUsingIllegalAtoms() : bool
ok() : bool
illegalAtoms() : std::vector<spot::formula>
parserErrors() : std::vector<ParserError>
setIllegalAtoms(std::vector<spot::formula>&& illegalAtoms) : void
operators
operator!() : bool
operator*() : spot :: formula&
friends
parseRtl(const std::string_view rtl, const spot::atomic_prop_set& allowedAtomicPropositions): RtlParsingResult
static functions
notLtlFormula(spot::formula&& rtl) : RtlParsingResult
ok(spot::formula&& rtl) : RtlParsingResult
withParserErrors(std::vector<ParserError>&& parserErrors) : RtlParsingResult

Figure 8.1: `RtlParsingResult` class

8.2 RTL Formula Utilities and Constants: the `formula` Component

The `formula` component provides a set of utilities and constants essential for constructing, manipulating, and evaluating temporal logic formulae. This component includes a variety of functions to handle the creation of atomic propositions, logical operators, and temporal operators, which are the building blocks of LTL formulae. In this section, we introduce the key utilities available in the `formula.h` header file, highlighting the functions used to build and manipulate LTL formulae. We will provide a detailed description of predefined constants like `sing` and `alive`, which are critical for RTL model checking and handling finite-time semantics.

8.2.1 Utilities and Operations on Formulae

The `formula.h` header file provides a set of utility functions for creating, modifying, and verifying properties of `spot::formula` objects. The following tables (Table 8.2, 8.3, 8.4, 8.5) summarise the available functions in `formula.h` for constructing and manipulating formulae:

Function
`top()` Returns the logical constant *true* formula (tt).
`bottom()` Returns the logical constant $false$ formula (ff).
`ap(std::string_view ap)` Converts a string into an atomic proposition formula.

Table 8.2: Basic logical constants and atomic propositions.

8.2.2 `RTL` Predefined Constants and Special Formulae

The module `formula_constants.h` defines several predefined constants and special formulae used in the context of model checking and temporal logic formulae. These constants and formulae are essential for handling special propositions, such as those related to time slicing, finite-time semantics, and the alternation between singular and open observables. As detailed in Section 4.3.3, the introduction of these special formulae, such as the alternation between singular and open observables, ensures that the `RTL` formulae can be appropriately converted into automata for model checking. These formulae introduce essential structural constraints to enforce the alternation between states representing singular and open intervals, a feature that is not naturally captured by the discretisation of the `RTL` formula. Therefore, we introduce these specialised `LTL` formulae to guarantee the correct interpretation of the system's behaviour.

Predefined Constants

- `g_sing`: A constant representing the *sing* atomic proposition. The *sing* atomic proposition is used to distinguish between open and singular intervals in the context of time slicing. It holds *true* in all and only the time points t_i of the time slicing τ, and $false$ elsewhere. This atomic proposition is essential because it helps address the model-checking problem of a `PolyhedralSystem` against

Function
`F(formula)` Creates an LTL "eventually" formula (F formula).
`X(formula)` Creates an LTL "next" formula (X formula).
`strongX(formula)` Creates an LTL strong "next" formula (X[!] formula).
`G(formula)` Creates an LTL "globally" formula (G formula).
`W(formula1, formula2)` Creates an LTL "weak until" formula (formula1 W formula2).
`U(formula1, formula2)` Creates an LTL "until" formula (formula1 U formula2).
`R(formula1, formula2)` Creates an LTL "release" formula (formula1 R formula2).
`Equiv(formula1, formula2)` Creates an LTL equivalence formula (formula1 formula2).

Table 8.3: Functions for creating temporal logic formulas.

Function
`And(formulae)` Creates a conjunction formula from a vector of sub-formulae.
`Or(formulae)` Creates a disjunction (Or) formula from a vector of sub-formulae.
`Not(formula)` Negates the given formula.
`sing()` Returns the *sing* atomic proposition.
`alive()` Returns the *alive* atomic proposition.
`AP(atoms)` Creates an atomic proposition set from a set of strings.

Table 8.4: Functions for conjunction, disjunction, negation, and atomic propositions.

Function
`isNotSing(formula)` Checks if the given formula is the negation of the *sing* atomic proposition.
`isSingOrNotSing(formula)` Checks if the given formula is either the *sing* atomic proposition or its negation.
`removeSing(formula)` Removes the *sing* atomic proposition from the given formula if present.
`containsSing(atomicPropositions)` Checks if the *sing* atomic proposition is present in the given set of atomic propositions.

Table 8.5: Functions for handling the *sing* atomic proposition.

Function
`collectAtomsNotIn(forbiddenAtoms, formula)` Collects all atomic propositions from the formula that are not in the provided forbidden set.
`collectPositiveLiterals(formula)` Collects all positive literals (non-negated atomic propositions) from the given formula.
`isCnf(formula)` Checks if the given formula is in Conjunctive Normal Form (CNF).
`collectCnfClauses(formula)` Collects all clauses from a Conjunctive Normal Form (CNF) formula.
`isConjunctionOfLiterals(formula)` Checks if the formula is a conjunction of literals.
`isNonRecurrent(formula)` Checks if the given formula is non-recurrent.

Table 8.6: Functions for collecting and checking properties of atomic propositions and formulas.

an `RTL` specification by reducing it to a decision problem for the discrete version of the logic, i.e. classic ω-regular LTL_ω and its finite variant LTL_f.

- `g_alive`: A constant representing the *alive* atomic proposition. The *alive* atomic proposition is used to embed the semantics of LTL_f into standard LTL_ω (which operates over infinite words). It is initially true but eventually becomes false forever, effectively marking the end of the finite trace. This allows Spot, designed for infinite-word semantics, to handle LTL_f formulae by rewriting them. For example, the LTL_f formula $(a \,\mathsf{U}\, b) \wedge \mathsf{F}\, c$ is transformed into the LTL_ω formula:

$$alive \wedge (a \,\mathsf{U}\, (alive \wedge b)) \;\wedge\; \mathsf{F}(alive \wedge c) \wedge (alive \,\mathsf{U}\, (\mathsf{G}\, \neg alive))$$

- `g_aliveUntilGNotAlive`: A constant representing the LTL formula $alive \,\mathsf{U}\, (\mathsf{G}\, \neg alive)$. This formula is key in reducing LTL_f (finite semantics) satisfiability to LTL_ω (on infinite traces) satisfiability. It guarantees that *alive* stays true until it fails and remains failed.

- `g_lastFinite`: A constant representing the atomic proposition *last* in the context of LTL_ω semantics. The *last* proposition is defined as $last \equiv \neg \mathsf{X}\, alive$, where *alive* is the atomic proposition marking the alive part of the trace. In the context of LTL_ω, *last* holds true at the final state of the finite trace.

Special Formulae The following formulae combine different temporal logic operations to model the alternation between singular and open observables, as well as finite-time semantics.

- `g_finiteAlternationSingOpenObservablesOneStep`: A constant representing the LTL formula that enforces alternation between singular and open observables and ensures finite-time semantics with a right-closed signal. This formula combines the following:
  - Alternation between singular and open observables: $\mathsf{G}(\neg alive \vee ((sing \leftrightarrow \mathsf{X}(alive \wedge \neg sing)) \vee \neg \mathsf{X}\, alive))$
  - Finite-time semantics: The trace must eventually end, marked by $\mathsf{F}(sing \wedge alive \wedge \neg \mathsf{X}\, alive)$

- `g_last`: A constant representing the atomic proposition *last* using the strong next operator. The *last* proposition is defined as $last \equiv \neg \mathsf{X}[!]\, true$, where $\mathsf{X}[!]$ is the strong next operator. This operator ensures that the next state exists and that the argument holds in that state. In the context of finite-time semantics, *last* holds true in the final state of the trace.

- `g_finiteAlternationSingOpenObservables`: A constant representing the LTL formula that enforces alternation between singular and open observables using the strong next operator and ensures finite-time semantics with a right-closed signal. This formula is similar to `g_finiteAlternationSingOpenObservablesOneStep`, but it uses the strong next operator $\mathsf{X}[!]$ instead of the weak next operator X. It combines:
  - Alternation between singular and open observables: $\mathsf{G}((\mathit{sing} \leftrightarrow \mathsf{X}[!]\neg \mathit{sing}) \vee \neg \mathsf{X}[!]\, \mathit{true})$
  - Finite-time semantics: The trace must eventually end, marked by $\mathsf{F}(\neg \mathsf{X}[!]\, \mathit{true} \wedge \mathit{sing})$
- `g_alternationSingOpenObservables`: A constant representing the LTL formula that enforces alternation between singular and open observables in the context of infinite-time semantics. The formula is:

  $$\mathsf{G}(\mathit{sing} \leftrightarrow \mathsf{X}\neg \mathit{sing})$$

  This formula ensures that the *sing* proposition alternates between true and false at each step of the trace.

8.2.3 Discretisation Transformation Rules

In addition to the discretisation rules defined in Section 4.3.2, we now define the discretisation rules for the operators W (weak until) and M (strong release). To define these new rules, we leverage the following two equivalences:

$$\begin{aligned} \varphi_1 \,\mathsf{W}\, \varphi_2 &\equiv \neg(\neg \varphi_1 \,\mathsf{U}\, \neg \varphi_2) \\ \varphi_1 \,\mathsf{M}\, \varphi_2 &\equiv \varphi_1 \,\mathsf{R}\, \varphi_2 \;\wedge\; \mathsf{F}(\varphi_1 \;\wedge\; \varphi_2) \end{aligned}$$

Using these equivalences, we can discretise the W and M operators by applying the discretisation rules for U, F, and R. Formally, the transformation $\mathtt{dsc}(\cdot)$ for these operators is defined as follows:

$$\begin{aligned} \mathtt{dsc}(\varphi_1 \,\mathsf{W}\, \varphi_2) &= \mathtt{dsc}(\neg(\neg \varphi_1 \,\mathsf{U}\, \neg \varphi_2)) \\ \mathtt{dsc}(\varphi_1 \,\mathsf{M}\, \varphi_2) &= \mathtt{dsc}(\varphi_1 \,\mathsf{R}\, \varphi_2) \;\wedge\; \mathtt{dsc}(\mathsf{F}(\varphi_1 \;\wedge\; \varphi_2)) \end{aligned}$$

All transformations are implemented in the `discretisation` module, within the header file `discretisation.h`. To apply the discretisation rules to an `RTL` formula, there are two classes available:

- `DiscreteFiniteLtlFormula`: This class represents an $\mathtt{RTL}_\mathrm{f}$ formula to which the discretisation rules have been applied. After applying these rules, the resulting $\mathtt{LTL}_\mathrm{f}$ formula is conjoined with the constant `g_finiteAlternationSingOpenObservables` (see Section 8.2.2).

- `DiscreteLtlFormula`: This class represents an $\texttt{RTL}_{\mathrm{f}}$ or $\texttt{RTL}_{\omega}$ formula to which the discretisation rules have been applied. If the original formula was an $\texttt{RTL}_{\mathrm{f}}$ formula, this class represents the corresponding $\texttt{LTL}_{\omega}$ formula, which is obtained by applying the process described in Section 6.3.7. If the formula was an $\texttt{RTL}_{\omega}$ formula, the class represents the $\texttt{LTL}_{\omega}$ formula obtained by discretising the original $\texttt{RTL}_{\omega}$ formula. The resulting $\texttt{LTL}_{\omega}$ formula is then conjoined with the constant `g_alternationSingOpenObservables` (see Section 8.2.2).

Figure 8.2 illustrates the two classes. To facilitate the process of discretisation, we define specific methods within the classes `DiscreteLtlFormula` and `DiscreteFiniteLtlFormula`. These methods are responsible for converting $\texttt{RTL}_{\omega}/\texttt{RTL}_{\mathrm{f}}$ formulae into their corresponding $\texttt{LTL}_{\omega}/\texttt{LTL}_{\mathrm{f}}$ representations. The Table 8.7 summarises the key methods available in these classes, with a focus on their factory methods for transforming $\texttt{RTL}_{\omega}/\texttt{RTL}_{\mathrm{f}}$ formulae into $\texttt{LTL}_{\omega}/\texttt{LTL}_{\mathrm{f}}$.

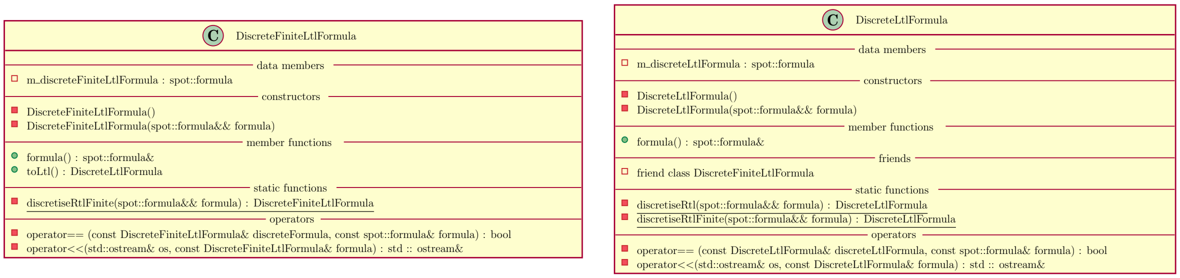

C
DiscreteFiniteLtlFormula
data members
m_discreteFiniteLtlFormula : spot::formula
constructors
DiscreteFiniteLtlFormula()
DiscreteFiniteLtlFormula(spot::formula&& formula)
member functions
formula() : spot::formula&
toLtl() : DiscreteLtlFormula
static functions
discretiseRtlFinite(spot::formula&& formula) : DiscreteFiniteLtlFormula
operators
operator== (const DiscreteFiniteLtlFormula& discreteFormula, const spot::formula& formula) : bool
operator<<(std::ostream& os, const DiscreteFiniteLtlFormula& formula) : std :: ostream&
C
DiscreteLtlFormula
data members
m_discreteLtlFormula : spot::formula
constructors
DiscreteLtlFormula()
DiscreteLtlFormula(spot::formula&& formula)
member functions
formula() : spot::formula&
friends
friend class DiscreteFiniteLtlFormula
static functions
discretiseRtl(spot::formula&& formula) : DiscreteLtlFormula
discretiseRtlFinite(spot::formula&& formula) : DiscreteLtlFormula
operators
operator== (const DiscreteLtlFormula& discreteLtlFormula, const spot::formula& formula) : bool
operator<<(std::ostream& os, const DiscreteLtlFormula& formula) : std :: ostream&

Figure 8.2: `DiscreteFiniteLtlFormula` and `DiscreteLtlFormula` classes

Class and Method	**Description**
`DiscreteLtlFormula::discretiseRtlFinite(formula)`	$\texttt{RTL}_{\mathrm{f}} \longrightarrow \texttt{LTL}_{\omega}$
`DiscreteLtlFormula::discretiseRtl(formula)`	$\texttt{RTL}_{\omega} \longrightarrow \texttt{LTL}_{\omega}$
`DiscreteFiniteLtlFormula::discretiseRtlFinite(formula)`	$\texttt{RTL}_{\mathrm{f}} \longrightarrow \texttt{LTL}_{\mathrm{f}}$
`DiscreteFiniteLtlFormula::toLtl()`	$\texttt{LTL}_{\mathrm{f}} \longrightarrow \texttt{LTL}_{\omega}$

Table 8.7: Main methods of `DiscreteLtlFormula` and `DiscreteFiniteLtlFormula` classes

8.2.4 Optimising Discretisation Process for Finite-Time Semantics

Table 8.7 shows two methods for converting an $\texttt{RTL}_f$ formula into an $\texttt{LTL}_f$ formula. The first method involves two steps: converting $\texttt{RTL}_f$ to $\texttt{LTL}_f$, and then further converting it into $\mathtt{LTL}_\omega$. This approach follows the sequence: $\mathtt{RTL}_f \longrightarrow \mathtt{LTL}_f \longrightarrow \mathtt{LTL}_\omega$, where the conversion from $\mathtt{LTL}_f$ to $\mathtt{LTL}_\omega$ is performed after the discretisation process using the function `spot::to_finite`. On the other hand, the more efficient method combines both transformations in one step. Instead of creating an intermediate $\mathtt{LTL}_f$ formula, the formula is directly discretised into $\mathtt{LTL}_\omega$, skipping the $\mathtt{LTL}_f$ stage. The optimized transformation proceeds as follows: $\mathtt{RTL}_f \longrightarrow \mathtt{LTL}_\omega$. This method is more efficient as it avoids unnecessary intermediate object creation and performs the transformation in a single pass. In both methods, the discretisation process also incorporates the `alive` atomic proposition. This proposition is essential for handling finite-time semantics, ensuring that the finite traces eventually end, and it is added during the conversion of $\mathtt{RTL}_f$ into $\mathtt{LTL}_\omega$. For properties over finite traces that require infinite-semantics treatment, the second method offers a significant performance advantage by consolidating both steps into a single operation. Consider the U (until) operator. The discretisation for the U operator proceeds as follows:

$$\mathtt{dsc}(\varphi_1 \,\mathsf{U}\, \varphi_2) = \mathtt{dsc}(\varphi_1) \,\mathsf{U}\, (\,\mathtt{dsc}(\varphi_2) \;\wedge\; (\,\mathtt{dsc}(\varphi_1) \;\vee\; \mathit{sing}) \;\wedge\; \mathit{alive})$$

This transformation illustrates how the U operator is handled during discretisation, ensuring that the necessary conditions for finite-time semantics are incorporated. The same reasoning applies to the other temporal operators, which follow the rules defined in [19].

—9—

Automata Are All You Need

CONTENTS: **9.1 The Base Abstract Class for Automata.** 9.1.1 Interface Design and Core Operations – 9.1.2 Acceptance Conditions. **9.2 Polyhedral Automata for LTL.** 9.2.1 From Edge-Based Denotation to State-Based Denotation – 9.2.2 Computing State Denotations with Don't-Care Observables – 9.2.3 Preprocessing the Transition-Based Automaton – 9.2.4 Postprocessing the State-Based Automaton – 9.2.5 Concrete Polyhedral Automata: Büchi, NFA, and Backward NFA. **9.3 Polyhedral Abstractions.** 9.3.1 Omnidirectional Polyhedral Abstraction. **9.4 Polyhedral Synchronous Product Automaton.** **9.5 Automata Emptiness Check Algorithms.** **9.6 On-the-Fly Algorithm for Finite-Time Semantics.**

Automata are the core computational model used throughout this tool to represent temporal logic formulae, polyhedral abstractions, and their interactions. This chapter introduces the foundational class hierarchy for automata and describes how formula automata and system abstractions are constructed, transformed, and composed. By translating `RTL` specifications into automata, we reduce the verification problem to a sequence of operations over graph-based structures.

9.1 The Base Abstract Class for Automata

All automata extend the abstract class `Automaton` shown in Figure 9.1. The `Automaton` class uses Spot's internal representations for graphs, i.e. `spot::twa_graph`, and provides an interface for manipulating and inspecting these graphs. Derived classes can implement specific behaviors. The base class also allows for easy integration with other components in the system that rely on automata-based analysis.

9.1.1 Interface Design and Core Operations

The `Automaton` class defines essential methods for accessing and manipulating the automaton's states, edges, and transition graphs. It also provides mechanisms for

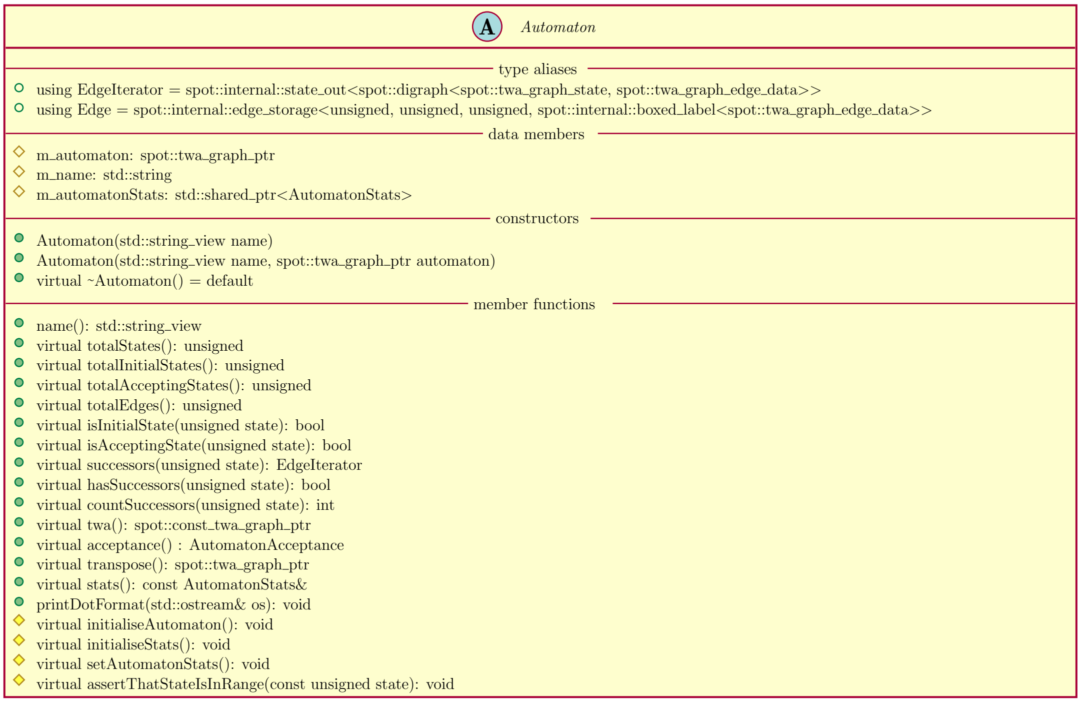

A
Automaton
type aliases
using EdgeIterator = spot::internal::state_out<spot::digraph<spot::twa_graph_state, spot::twa_graph_edge_data>>
using Edge = spot::internal::edge_storage<unsigned, unsigned, unsigned, spot::internal::boxed_label<spot::twa_graph_edge_data>>
data members
m_automaton: spot::twa_graph_ptr
m_name: std::string
m_automatonStats: std::shared_ptr<AutomatonStats>
constructors
Automaton(std::string_view name)
Automaton(std::string_view name, spot::twa_graph_ptr automaton)
virtual ~Automaton() = default
member functions
name(): std::string_view
virtual totalStates(): unsigned
virtual totalInitialStates(): unsigned
virtual totalAcceptingStates(): unsigned
virtual totalEdges(): unsigned
virtual isInitialState(unsigned state): bool
virtual isAcceptingState(unsigned state): bool
virtual successors(unsigned state): EdgeIterator
virtual hasSuccessors(unsigned state): bool
virtual countSuccessors(unsigned state): int
virtual twa(): spot::const_twa_graph_ptr
virtual acceptance() : AutomatonAcceptance
virtual transpose(): spot::twa_graph_ptr
virtual stats(): const AutomatonStats&
printDotFormat(std::ostream& os): void
virtual initialiseAutomaton(): void
virtual initialiseStats(): void
virtual setAutomatonStats(): void
virtual assertThatStateIsInRange(const unsigned state): void

Figure 9.1: `Automaton` class

get the acceptance condition and get automaton statistics. The Table 9.1 show a summary of the key public methods available in this class.

9.1.2 Acceptance Conditions

An `Automaton` can have different acceptance conditions. Currently, three acceptance conditions are supported: NFA, Büchi, and an acceptance condition where all states are accepting. These conditions are represented by the `AutomatonAcceptance` class, as shown in Figure 9.2. However, it is possible to add new variants to this enumeration to reflect other acceptance conditions supported by Spot. This is because all automata in the system are essentially Spot automata internally.

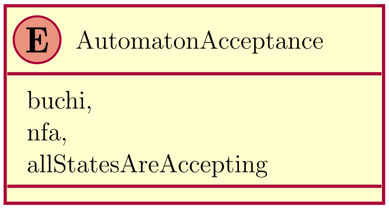

E
AutomatonAcceptance
buchi,
nfa,
allStatesAreAccepting

Figure 9.2: `AutomatonAcceptance` class

Method	**Description**
`name()`	Returns the automaton's name (identifier).
`totalStates()`	Returns the total number of states in the automaton.
`totalInitialStates()`	Returns the number of initial states.
`totalAcceptingStates()`	Returns the number of accepting states.
`totalEdges()`	Returns the total number of edges in the automaton.
`isInitialState(unsigned state)`	Checks if a given state is an initial state.
`isAcceptingState(unsigned state)`	Checks if a given state is an accepting state.
`successors(unsigned state)`	Retrieves the successor iterator for a given state.
`hasSuccessors(unsigned state)`	Checks if a state has any successors.
`countSuccessors(unsigned state)`	Counts the number of outgoing edges from a state.
`twa()`	Returns a shared pointer to the automaton's transition graph.
`transpose()`	Returns a new automaton with reversed edges (transposed).
`acceptance()`	Retrieves the automaton's acceptance condition.
`stats()`	Returns the automaton's statistics.
`printDotFormat(std::ostream& os)`	Outputs the automaton in Graphviz DOT format.

Table 9.1: Public methods of the `Automaton` class.

9.2 Polyhedral Automata for LTL

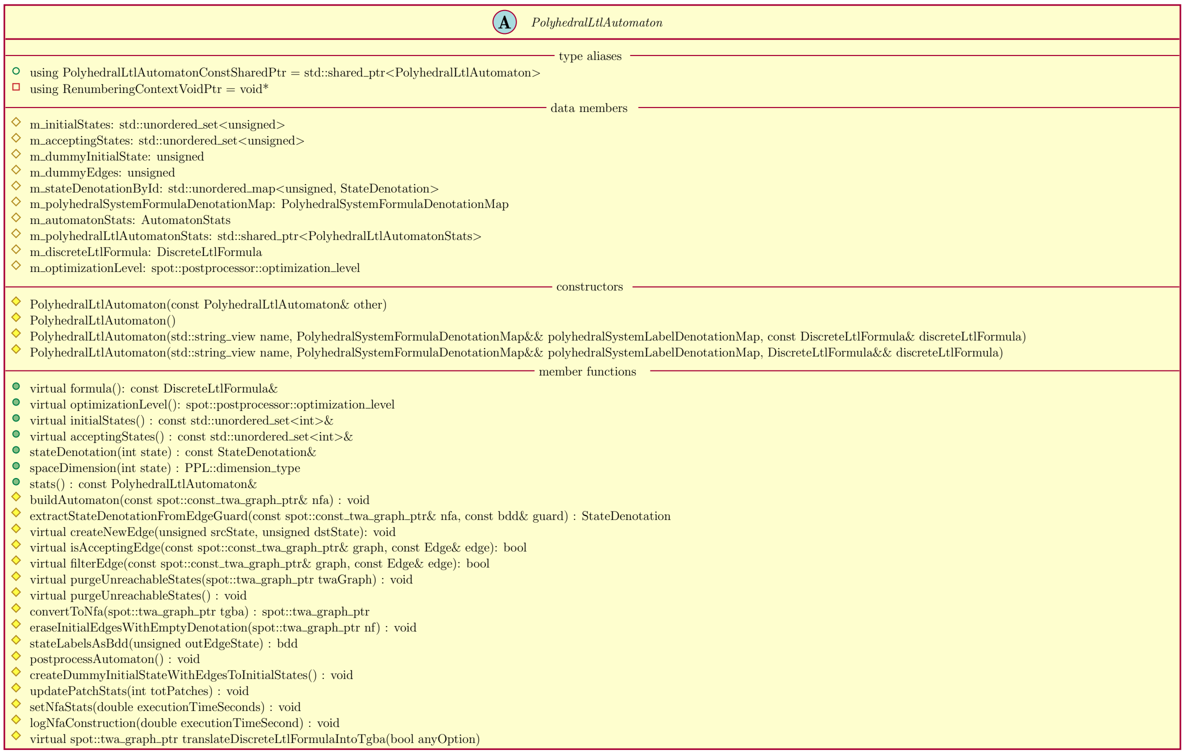
The automaton of the discretised `RTL` formula, which assigns to each state $s \in S$ a subset of $AP \cup \{sing\}$ along with its denotation $[\![s]\!]$ (see Section 4.3.3), is represented by the abstract class `PolyhedralLtlAutomaton`. This class takes the discretised formula `DiscreteLtlFormula`, which represents an $\texttt{LTL}_\omega$ formula, and uses Spot to translate the formula into an automaton (see Section 6.3.6). Starting from this automaton, which is essentially a transition-based ω-automaton (see Section 6.3.4) with Büchi acceptance, the final automaton with state labels is constructed. It will always be interpreted as a state-based automaton. As an abstract class, the specific post-translation operations for the formula are determined by the various implementations, which will be discussed in Section 9.2.5. However, some operations are common to all implementations, so we will begin by covering these. The `PolyhedralLtlAutomaton` class is shown in Figure 9.3, and it inherits from the `Automaton` class, adding several new methods tailored to the specific needs of polyhedral automata. These methods allow access to the automaton's initial and accepting states, retrieve state denotations, and provide information on the formula and its Spot's translation optimisation level, as well as the space dimension. The Table 9.2 summarises the public methods of `PolyhedralLtlAutomaton`:

Method	**Description**
`initialStates()`	Returns the set of initial states in the automaton.
`acceptingStates()`	Returns the set of accepting states in the automaton.
`stateDenotation(unsigned state)`	Retrieves the denotation of a given state.
`formula()`	Returns the discretised $\texttt{LTL}_\omega$ formula.
`optimizationLevel()`	Returns the optimization level for the automaton.
`spaceDimension()`	Returns the dimension of the polyhedral space.

Table 9.2: Methods of the `PolyhedralLtlAutomaton` class.

9.2.1 From Edge-Based Denotation to State-Based Denotation

As discussed in Section 6.3.6, the translation of an $\texttt{LTL}_\omega$ formula into an automaton is fully handled by Spot through its `spot::translator` class. The resulting automaton is a transition-based ω-automaton, which, in this case, uses Büchi acceptance. As detailed in Section 6.3.5, Spot labels the automaton's edges with `spot::formula` objects. In this framework, the automaton's transitions are not labelled with individual letters, but rather with boolean formulae, which can be viewed as a way of aggregating



```
A PolyhedralLtlAutomaton

type aliases
○ using PolyhedralLtlAutomatonConstSharedPtr = std::shared_ptr<PolyhedralLtlAutomaton>
□ using RenumberingContextVoidPtr = void*

data members
◇ m_initialStates: std::unordered_set<unsigned>
◇ m_acceptingStates: std::unordered_set<unsigned>
◇ m_dummyInitialState: unsigned
◇ m_dummyEdges: unsigned
◇ m_stateDenotationById: std::unordered_map<unsigned, StateDenotation>
◇ m_polyhedralSystemFormulaDenotationMap: PolyhedralSystemFormulaDenotationMap
◇ m_automatonStats: AutomatonStats
◇ m_polyhedralLtlAutomatonStats: std::shared_ptr<PolyhedralLtlAutomatonStats>
◇ m_discreteLtlFormula: DiscreteLtlFormula
◇ m_optimizationLevel: spot::postprocessor::optimization_level

constructors
◇ PolyhedralLtlAutomaton(const PolyhedralLtlAutomaton& other)
◇ PolyhedralLtlAutomaton()
◇ PolyhedralLtlAutomaton(std::string_view name, PolyhedralSystemFormulaDenotationMap&& polyhedralSystemLabelDenotationMap, const DiscreteLtlFormula& discreteLtlFormula)
◇ PolyhedralLtlAutomaton(std::string_view name, PolyhedralSystemFormulaDenotationMap&& polyhedralSystemLabelDenotationMap, DiscreteLtlFormula&& discreteLtlFormula)

member functions
● virtual formula(): const DiscreteLtlFormula&
● virtual optimizationLevel(): spot::postprocessor::optimization_level
● virtual initialStates() : const std::unordered_set<int>&
● virtual acceptingStates() : const std::unordered_set<int>&
● stateDenotation(int state) : const StateDenotation&
● spaceDimension(int state) : PPL::dimension_type
● stats() : const PolyhedralLtlAutomaton&
◇ buildAutomaton(const spot::const_twa_graph_ptr& nfa) : void
◇ extractStateDenotationFromEdgeGuard(const spot::const_twa_graph_ptr& nfa, const bdd& guard) : StateDenotation
◇ virtual createNewEdge(unsigned srcState, unsigned dstState): void
◇ virtual isAcceptingEdge(const spot::const_twa_graph_ptr& graph, const Edge& edge): bool
◇ virtual filterEdge(const spot::const_twa_graph_ptr& graph, const Edge& edge): bool
◇ virtual purgeUnreachableStates(spot::twa_graph_ptr twaGraph) : void
◇ virtual purgeUnreachableStates() : void
◇ convertToNfa(spot::twa_graph_ptr tgba) : spot::twa_graph_ptr
◇ eraseInitialEdgesWithEmptyDenotation(spot::twa_graph_ptr nf) : void
◇ stateLabelsAsBdd(unsigned outEdgeState) : bdd
◇ postprocessAutomaton() : void
◇ createDummyInitialStateWithEdgesToInitialStates() : void
◇ updatePatchStats(int totPatches) : void
◇ setNfaStats(double executionTimeSeconds) : void
◇ logNfaConstruction(double executionTimeSecond) : void
◇ virtual spot::twa_graph_ptr translateDiscreteLtlFormulaIntoTgba(bool anyOption)
```

Figure 9.3: `PolyhedralLtlAutomaton` class

several letter-labelled transitions into a single one. Since the labelling function λ labels the states, and we want each state $s \in S$ to have its own denotation $[\![s]\!]$, a new automaton must be created after the translation. This new automaton will be the final one, built from the $\texttt{LTL}_\omega$ automaton produced by Spot. The goal is to transform each transition into a state and connect these newly created states with transitions if the original transitions associated with them are consecutive (i.e. the destination state of one transition is the same as the source state of the next). All the procedure is handled by protected virtual method `buildAutomaton()` present in `PolyhedralLtlAutomaton`. Formally, if $\mathcal{A} = (S, S_0, \delta, T_F)$ is a transition-based automaton where T_F is the set of accepting transitions, we define the new state-based automaton $\mathcal{A}' = (S', S_0', \delta', \lambda, S_F')$ as follows:

- $S' = \{s_t \mid t \in \delta\}$, where each s_t represents a new state derived from a transition t,
- $S_0' = \{s_t \mid t = (s, a, s')$ and $s \in S_0\}$, the set of initial states in S' corresponding to initial transitions in the original automaton,
- $\delta' \subseteq S' \times S'$ is defined by the following rule:

  $$\frac{t = (s, a, s') \wedge t' = (s', b, s'')}{s_t \longrightarrow s_t'}$$

  This rule ensures that there is a transition between states in S' if the original transitions were consecutive.
- $\lambda(s_t) = a$, where $t = (s, a, s')$ and a is the label of the transition,
- $S_F' = \{s_t \mid t \in T_F\} \subseteq S'$, the set of accepting states in the new automaton.

9.2.2 Computing State Denotations with Don't-Care Observables

As described in Section 9.2.1, each transition in the automaton is transformed into a state. Each newly created state must be associated with a denotation, which is represented by the `StateDenotation` class shown in Figure 9.4. This class represents the denotation of a state and is immutable. The mapping between automaton states and `StateDenotation` objects is maintained within the automaton itself, specifically within the member variable `m_stateDenotationById` of the `PolyhedralLtlAutomaton` class. A state's denotation is derived from the formula associated with the transition that generated the state. An instance of `StateDenotation` is marked as singular if the atomic proposition $sing$ is present in the formula. This formula is always in Conjunctive Normal Form (CNF), meaning it is a disjunction of clauses. A CNF formula of this type can represent multiple observables, as some atomic propositions can be treated as don't-care. Therefore, given a CNF formula, the state's denotation is computed by performing the union of the intersections of the interpretations of the

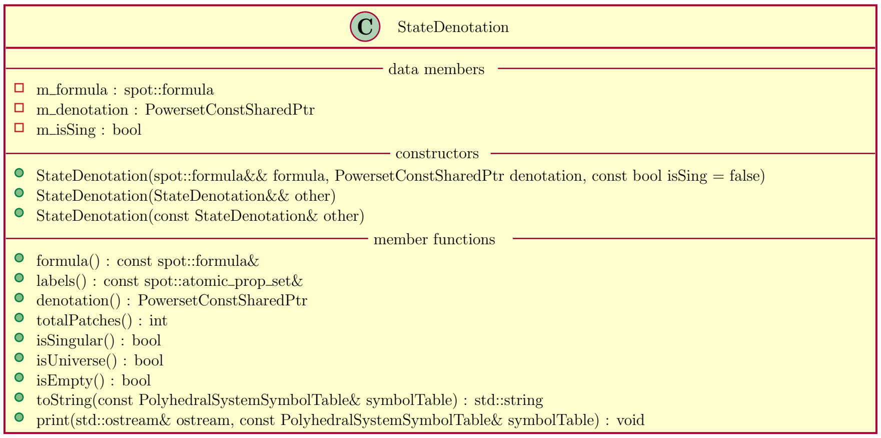

C StateDenotation
data members
□ m_formula : spot::formula
□ m_denotation : PowersetConstSharedPtr
□ m_isSing : bool
constructors
● StateDenotation(spot::formula&& formula, PowersetConstSharedPtr denotation, const bool isSing = false)
● StateDenotation(StateDenotation&& other)
● StateDenotation(const StateDenotation& other)
member functions
● formula() : const spot::formula&
● labels() : const spot::atomic_prop_set&
● denotation() : PowersetConstSharedPtr
● totalPatches() : int
● isSingular() : bool
● isUniverse() : bool
● isEmpty() : bool
● toString(const PolyhedralSystemSymbolTable& symbolTable) : std::string
● print(std::ostream& ostream, const PolyhedralSystemSymbolTable& symbolTable) : void

Figure 9.4: `StateDenotation` class

atomic propositions. If an atomic proposition is negated, its negated interpretation is taken into account. The `PolyhedralSystemFormulaDenotationMap` class, shown in Figure 9.5, calculates a `Powerset` from a CNF `spot::formula` by executing exactly what was described above. The `PolyhedralSystemFormulaDenotationMap` class holds a reference to the `PolyhedralSystem` to retrieve the interpretations of the atomic propositions. It also stores all the computed results to avoid recomputing the same denotation multiple times. For this reason, the main method is named `getOrComputeDenotation`, which either retrieves the precomputed denotation or computes it if necessary.

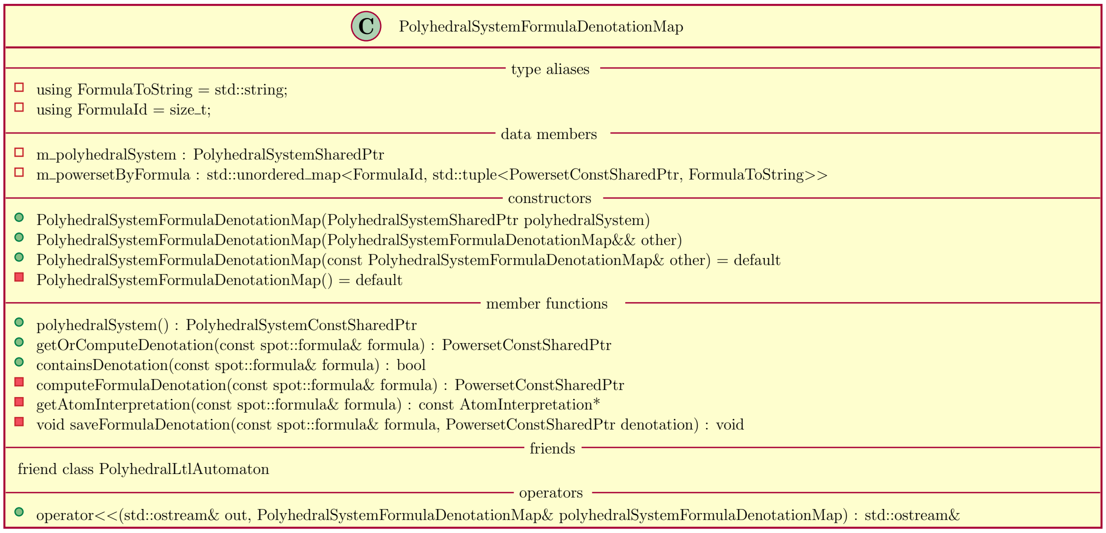

C PolyhedralSystemFormulaDenotationMap
type aliases
□ using FormulaToString = std::string;
□ using FormulaId = size_t;
data members
□ m_polyhedralSystem : PolyhedralSystemSharedPtr
□ m_powersetByFormula : std::unordered_map<FormulaId, std::tuple<PowersetConstSharedPtr, FormulaToString>>
constructors
● PolyhedralSystemFormulaDenotationMap(PolyhedralSystemSharedPtr polyhedralSystem)
● PolyhedralSystemFormulaDenotationMap(PolyhedralSystemFormulaDenotationMap&& other)
● PolyhedralSystemFormulaDenotationMap(const PolyhedralSystemFormulaDenotationMap& other) = default
■ PolyhedralSystemFormulaDenotationMap() = default
member functions
● polyhedralSystem() : PolyhedralSystemConstSharedPtr
● getOrComputeDenotation(const spot::formula& formula) : PowersetConstSharedPtr
● containsDenotation(const spot::formula& formula) : bool
■ computeFormulaDenotation(const spot::formula& formula) : PowersetConstSharedPtr
■ getAtomInterpretation(const spot::formula& formula) : const AtomInterpretation*
■ void saveFormulaDenotation(const spot::formula& formula, PowersetConstSharedPtr denotation) : void
friends
friend class PolyhedralLtlAutomaton
operators
● operator<<(std::ostream& out, PolyhedralSystemFormulaDenotationMap& polyhedralSystemFormulaDenotationMap) : std::ostream&

Figure 9.5: `PolyhedralSystemFormulaDenotationMap` class

9.2.3 Preprocessing the Transition-Based Automaton

Regardless of the specific implementation of `PolyhedralLtlAutomaton`, before transforming the transition-based automaton into a state-based automaton with state labels and denotations, it is beneficial to preprocess the automaton to get a more compact representation. Specifically, before calling the method `PolyhedralLtlAutomaton::buildAutomaton`, outgoing edges from initial states that are labelled with formulae resulting in empty denotations are removed. Subsequently, all unreachable states and edges are removed from the automaton. This is achieved by executing the following two methods before performing the transition-based to state-based translation process:

1. `PolyhedralLtlAutomaton::eraseInitialEdgesWithEmptyDenotation`: Removes outgoing edges from initial states that are labelled with formulas producing empty denotations.
2. `PolyhedralLtlAutomaton::purgeUnreachableStates`: Removes unreachable states and edges.

9.2.4 Postprocessing the State-Based Automaton

Once the final automaton has been constructed, the virtual method `PolyhedralLtlAutomaton::postprocessAutomaton()` is invoked. This method can be overridden by the derived implementations, but its default implementation creates a single initial state with outgoing edges to the actual initial states of the automaton, which are obtained according to the procedure described in Section 9.2.1. This step is necessary because Spot allows only one initial state. However, externally, `PolyhedralLtlAutomaton` will always return the actual initial states via the `PolyhedralLtlAutomaton::initialStates()` method. The fake initial state and its outgoing edges are hidden from the outside (they are not even counted when invoking methods to retrieve the total number of states or edges). Next, the default implementation of `postprocessAutomaton()` removes all unreachable states and edges from the final automaton, as states derived from formulae with empty denotations are not created during the transition-based to state-based translation process.

9.2.5 Concrete Polyhedral Automata: Büchi, NFA, and Backward NFA

Figure 9.6 illustrates the different implementations of the `PolyhedralLtlAutomaton` class. Each implementation corresponds to a specific type of automaton, tailored to handle particular aspects of temporal logic formulae. The three main implementations are:

- `PolyhedralFiniteLtlAutomaton`: This automaton is generated by translating a discretised $\mathtt{RTL_f}$ formula into a Non-deterministic Finite Automaton (NFA), which is used for finite-state analysis.

- `PolyhedralBuchiLtlAutomaton`: This automaton is created by translating a discretised $\mathtt{RTL}_\omega$ formula into a Büchi automaton, making it suitable for handling infinite words, which is essential in many temporal logic applications.

- `BackwardNFA`: This class extends `PolyhedralFiniteLtlAutomaton` and provides a transposed version of the NFA. It is specifically designed for use in the on-the-fly algorithm, as described in Section 4.4.1.

Each concrete automaton exposes a factory method `buildAutomaton`, which is the only way to construct these automata.

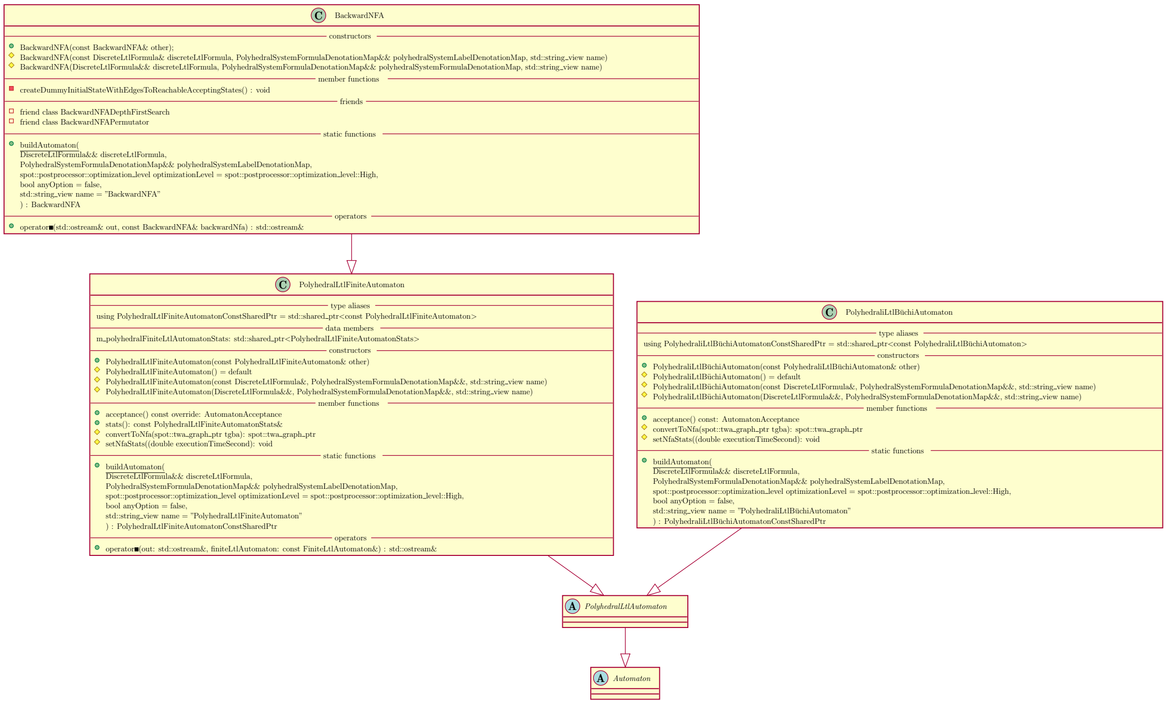

BackwardNFA
constructors
BackwardNFA(const BackwardNFA& other);
BackwardNFA(const DiscreteLtlFormula& discreteLtlFormula, PolyhedralSystemFormulaDenotationMap&& polyhedralSystemLabelDenotationMap, std::string_view name)
BackwardNFA(DiscreteLtlFormula&& discreteLtlFormula, PolyhedralSystemFormulaDenotationMap&& polyhedralSystemFormulaDenotationMap, std::string_view name)
member functions
createDummyInitialStateWithEdgesToReachableAcceptingStates() : void
friends
friend class BackwardNFADepthFirstSearch
friend class BackwardNFAPermutator
static functions
buildAutomaton(
DiscreteLtlFormula&& discreteLtlFormula,
PolyhedralSystemFormulaDenotationMap&& polyhedralSystemLabelDenotationMap,
spot::postprocessor::optimization_level optimizationLevel = spot::postprocessor::optimization_level::High,
bool anyOption = false,
std::string_view name = "BackwardNFA"
) : BackwardNFA
operators
operator■(std::ostream& out, const BackwardNFA& backwardNfa) : std::ostream&
PolyhedralLtlFiniteAutomaton
type aliases
using PolyhedralLtlFiniteAutomatonConstSharedPtr = std::shared_ptr<const PolyhedralLtlFiniteAutomaton>
data members
m_polyhedralFiniteLtlAutomatonStats: std::shared_ptr<PolyhedralLtlFiniteAutomatonStats>
constructors
PolyhedralLtlFiniteAutomaton(const PolyhedralLtlFiniteAutomaton& other)
PolyhedralLtlFiniteAutomaton() = default
PolyhedralLtlFiniteAutomaton(const DiscreteLtlFormula&, PolyhedralSystemFormulaDenotationMap&&, std::string_view name)
PolyhedralLtlFiniteAutomaton(DiscreteLtlFormula&&, PolyhedralSystemFormulaDenotationMap&&, std::string_view name)
member functions
acceptance() const override: AutomatonAcceptance
stats(): const PolyhedralLtlFiniteAutomatonStats&
convertToNfa(spot::twa_graph_ptr tgba): spot::twa_graph_ptr
setNfaStats((double executionTimeSecond): void
static functions
buildAutomaton(
DiscreteLtlFormula&& discreteLtlFormula,
PolyhedralSystemFormulaDenotationMap&& polyhedralSystemLabelDenotationMap,
spot::postprocessor::optimization_level optimizationLevel = spot::postprocessor::optimization_level::High,
bool anyOption = false,
std::string_view name = "PolyhedralLtlFiniteAutomaton"
) : PolyhedralLtlFiniteAutomatonConstSharedPtr
operators
operator■(out: std::ostream&, finiteLtlAutomaton: const FiniteLtlAutomaton&) : std::ostream&
PolyhedraliLtlBüchiAutomaton
type aliases
using PolyhedraliLtlBüchiAutomatonConstSharedPtr = std::shared_ptr<const PolyhedraliLtlBüchiAutomaton>
constructors
PolyhedraliLtlBüchiAutomaton(const PolyhedraliLtlBüchiAutomaton& other)
PolyhedraliLtlBüchiAutomaton() = default
PolyhedraliLtlBüchiAutomaton(const DiscreteLtlFormula&, PolyhedralSystemFormulaDenotationMap&&, std::string_view name)
PolyhedraliLtlBüchiAutomaton(DiscreteLtlFormula&&, PolyhedralSystemFormulaDenotationMap&&, std::string_view name)
member functions
acceptance() const: AutomatonAcceptance
convertToNfa(spot::twa_graph_ptr tgba): spot::twa_graph_ptr
setNfaStats((double executionTimeSecond): void
static functions
buildAutomaton(
DiscreteLtlFormula&& discreteLtlFormula,
PolyhedralSystemFormulaDenotationMap&& polyhedralSystemLabelDenotationMap,
spot::postprocessor::optimization_level optimizationLevel = spot::postprocessor::optimization_level::High,
bool anyOption = false,
std::string_view name = "PolyhedraliLtlBüchiAutomaton"
) : PolyhedraliLtlBüchiAutomatonConstSharedPtr
PolyhedralLtlAutomaton
Automaton

Figure 9.6: Concrete Polyhedral Automata Classes

9.3 Polyhedral Abstractions

A polyhedral abstraction, as described in Section 9.3, is represented by the abstract class `PolyhedralAbstraction`, shown in Figure 9.7. The `PolyhedralAbstraction` class extends the `Automaton` class, treating it as an automaton with an `AutomatonAcceptance` condition set to `allStatesAreAccepting` (see Section 9.1.2). This design choice simplifies the synchronous product between a polyhedral abstraction and a formula automaton (see Section 9.4). In this section, we will focus on the implementation of the omnidirectional polyhedral abstraction described in Section 4.4.2. The

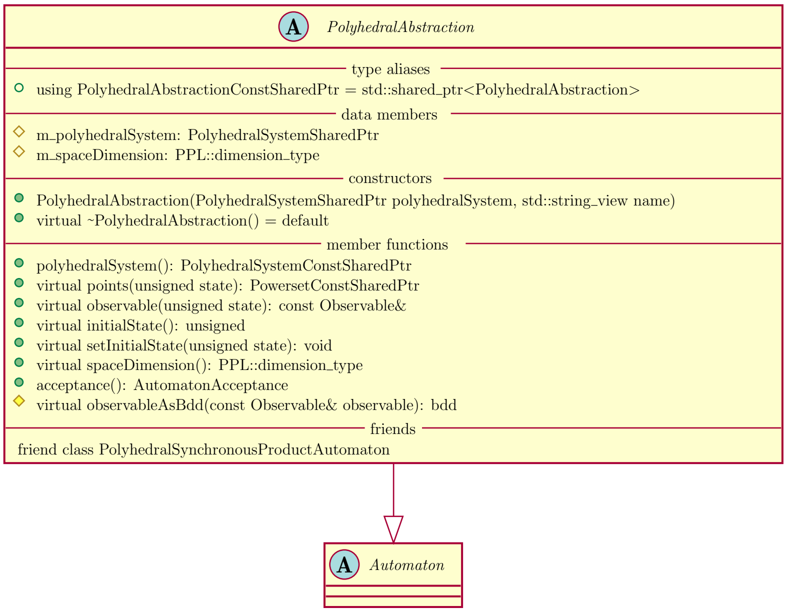

A
PolyhedralAbstraction
type aliases
using PolyhedralAbstractionConstSharedPtr = std::shared_ptr<PolyhedralAbstraction>
data members
m_polyhedralSystem: PolyhedralSystemSharedPtr
m_spaceDimension: PPL::dimension_type
constructors
PolyhedralAbstraction(PolyhedralSystemSharedPtr polyhedralSystem, std::string_view name)
virtual ~PolyhedralAbstraction() = default
member functions
polyhedralSystem(): PolyhedralSystemConstSharedPtr
virtual points(unsigned state): PowersetConstSharedPtr
virtual observable(unsigned state): const Observable&
virtual initialState(): unsigned
virtual setInitialState(unsigned state): void
virtual spaceDimension(): PPL::dimension_type
acceptance(): AutomatonAcceptance
virtual observableAsBdd(const Observable& observable): bdd
friends
friend class PolyhedralSynchronousProductAutomaton
A
Automaton

Figure 9.7: `PolyhedralAbstraction` class

general polyhedral abstraction, outlined earlier in Section 4.4.1, is impractical for direct use due to its complexity, making its construction unfeasible in many cases. However, an implementation of this abstraction does exist and is available within the `abstractions` component.

9.3.1 Omnidirectional Polyhedral Abstraction

A class diagram of the `abstractions/omnidirectional` component is shown in Figure 9.8. As described in Section 4.5.1, a key concept in the omnidirectional abstraction is the notion of a tile. The `Tile` class represents a tile of an observable. A state (P) with a single `Tile` is represented by the `TileNode` class, while a state (P, Q, R) with three `Tiles` is represented by the `TripleTileNode` class. The extraction of tiles is handled by the abstract class `TileExtractor`. Two implementations of `TileExtractor` exist:

- `TileExtractorGraph`: This implementation uses a graph-based approach to extract tiles. The method creates an undirected graph where:
  - Nodes represent individual patches (all `Poly` objects contained in the interpretation of the `Observable`).
  - An edge exists between two nodes if and only if the corresponding `Polys` are adjacent (see Section 2.1.2).

  It identifies connected components in the graph, with each connected component representing a `Tile`.

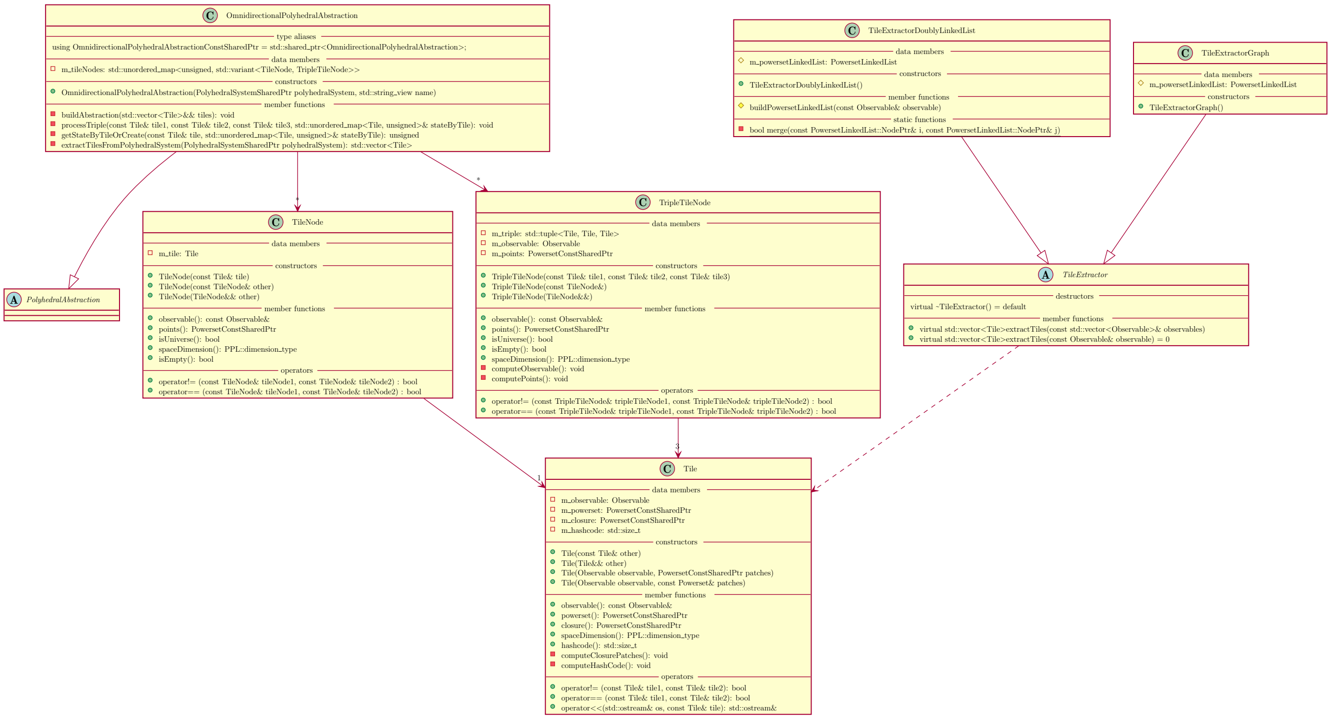

OmnidirectionalPolyhedralAbstraction
type aliases
using OmnidirectionalPolyhedralAbstractionConstSharedPtr = std::shared_ptr<OmnidirectionalPolyhedralAbstraction>;
data members
m_tileNodes: std::unordered_map<unsigned, std::variant<TileNode, TripleTileNode>>
constructors
OmnidirectionalPolyhedralAbstraction(PolyhedralSystemSharedPtr polyhedralSystem, std::string_view name)
member functions
buildAbstraction(std::vector<Tile>&& tiles): void
processTriple(const Tile& tile1, const Tile& tile2, const Tile& tile3, std::unordered_map<Tile, unsigned>& stateByTile): void
getStateByTileOrCreate(const Tile& tile, std::unordered_map<Tile, unsigned>& stateByTile): unsigned
extractTilesFromPolyhedralSystem(PolyhedralSystemSharedPtr polyhedralSystem): std::vector<Tile>
TileExtractorDoublyLinkedList
data members
m_powersetLinkedList: PowersetLinkedList
constructors
TileExtractorDoublyLinkedList()
member functions
buildPowersetLinkedList(const Observable& observable)
static functions
bool merge(const PowersetLinkedList::NodePtr& i, const PowersetLinkedList::NodePtr& j)
TileExtractorGraph
data members
m_powersetLinkedList: PowersetLinkedList
constructors
TileExtractorGraph()
*
*
PolyhedralAbstraction
TileNode
data members
m_tile: Tile
constructors
TileNode(const Tile& tile)
TileNode(const TileNode& other)
TileNode(TileNode&& other)
member functions
observable(): const Observable&
points(): PowersetConstSharedPtr
isUniverse(): bool
spaceDimension(): PPL::dimension_type
isEmpty(): bool
operators
operator!= (const TileNode& tileNode1, const TileNode& tileNode2) : bool
operator== (const TileNode& tileNode1, const TileNode& tileNode2) : bool
TripleTileNode
data members
m_triple: std::tuple<Tile, Tile, Tile>
m_observable: Observable
m_points: PowersetConstSharedPtr
constructors
TripleTileNode(const Tile& tile1, const Tile& tile2, const Tile& tile3)
TripleTileNode(const TileNode&)
TripleTileNode(TileNode&&)
member functions
observable(): const Observable&
points(): PowersetConstSharedPtr
isUniverse(): bool
isEmpty(): bool
spaceDimension(): PPL::dimension_type
computeObservable(): void
computePoints(): void
operators
operator!= (const TripleTileNode& tripleTileNode1, const TripleTileNode& tripleTileNode2) : bool
operator== (const TripleTileNode& tripleTileNode1, const TripleTileNode& tripleTileNode2) : bool
TileExtractor
destructors
virtual ~TileExtractor() = default
member functions
virtual std::vector<Tile>extractTiles(const std::vector<Observable>& observables)
virtual std::vector<Tile>extractTiles(const Observable& observable) = 0
3
1
Tile
data members
m_observable: Observable
m_powerset: PowersetConstSharedPtr
m_closure: PowersetConstSharedPtr
m_hashcode: std::size_t
constructors
Tile(const Tile& other)
Tile(Tile&& other)
Tile(Observable observable, PowersetConstSharedPtr patches)
Tile(Observable observable, const Powerset& patches)
member functions
observable(): const Observable&
powerset(): PowersetConstSharedPtr
closure(): PowersetConstSharedPtr
spaceDimension(): PPL::dimension_type
hashcode(): std::size_t
computeClosurePatches(): void
computeHashCode(): void
operators
operator!= (const Tile& tile1, const Tile& tile2): bool
operator== (const Tile& tile1, const Tile& tile2): bool
operator<<(std::ostream& os, const Tile& tile): std::ostream&

Figure 9.8: `omnidirectional` component class diagram

- `TileExtractorDoublyLinkedList`: This implementation uses a doubly linked list to extract tiles. The key advantage of this approach is that it efficiently handles the merging of adjacent patches by maintaining a linked list of powersets. The implementation follows these steps:

  - It first builds the initial doubly linked list from the `Observable`'s patches, where each node represents a powerset of size 1.
  - It then merges patches that are geometrically adjacent by linking them in the list, forming a `Tile`.

  This approach is particularly useful for scenarios where memory efficiency and fast merging of adjacent patches are critical.

Both implementations provide methods for extracting tiles from a single `Observable` or a collection of observables. These methods return a vector of `Tiles`, which can then be used in further computations within the omnidirectional polyhedral abstraction. While both implementations are valid, the graph-based implementation (`TileExtractorGraph`) was chosen for its better practical time efficiency.

9.4 Polyhedral Synchronous Product Automaton

The synchronous product between the formula automaton and a polyhedral abstraction is represented by the `PolyhedralSynchronousProductAutomaton` class, as shown in Figure 9.9. The `PolyhedralSynchronousProductAutomaton` class holds references to an instance of `PolyhedralLtlAutomaton` and an instance of `PolyhedralAbstraction`. Let $\mathcal{A} = (2^{\widehat{AP}}, S, \delta, \lambda, S_0, S_F)$ be the automaton of an `RTL` formula, where $2^{\widehat{AP}} = AP \cup \{sing\}$, and let $G = (V, E, pts, obs)$ be a polyhedral abstraction, where both $\mathcal{A}$ and G are defined on the same polyhedral system. We define the synchronous product $\mathcal{A} \otimes G = (2^{\widehat{AP}}, S', \delta', S_0', S_F', pts')$ as follows:

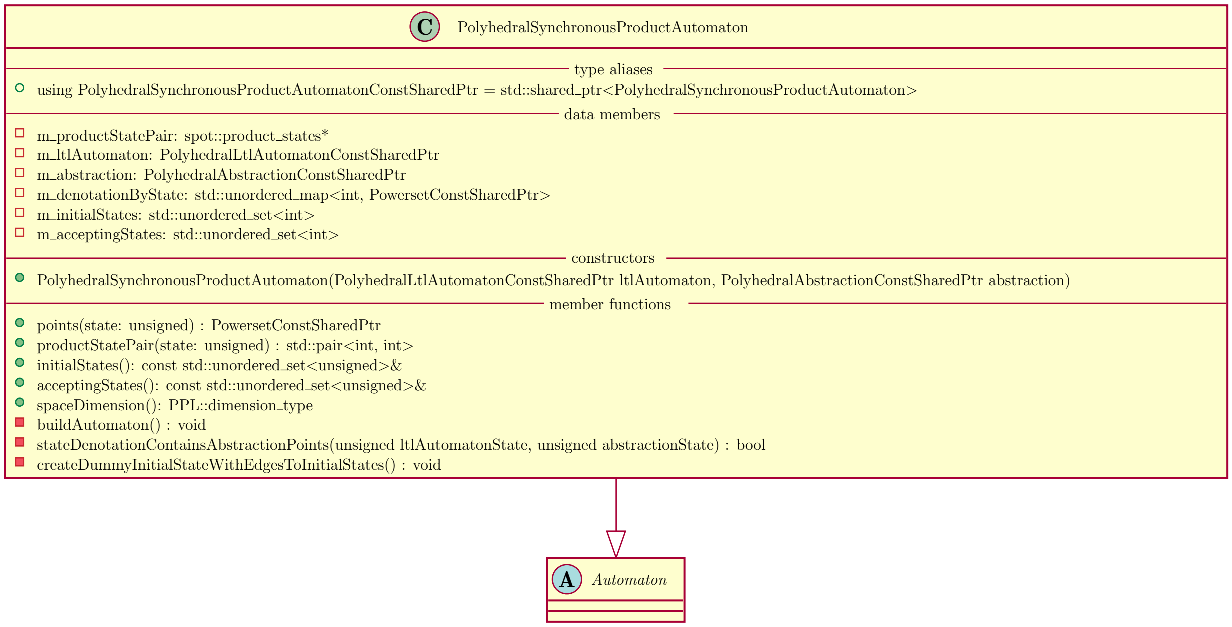

C
PolyhedralSynchronousProductAutomaton
type aliases
○ using PolyhedralSynchronousProductAutomatonConstSharedPtr = std::shared_ptr<PolyhedralSynchronousProductAutomaton>
data members
☐ m_productStatePair: spot::product_states*
☐ m_ltlAutomaton: PolyhedralLtlAutomatonConstSharedPtr
☐ m_abstraction: PolyhedralAbstractionConstSharedPtr
☐ m_denotationByState: std::unordered_map<int, PowersetConstSharedPtr>
☐ m_initialStates: std::unordered_set<int>
☐ m_acceptingStates: std::unordered_set<int>
constructors
● PolyhedralSynchronousProductAutomaton(PolyhedralLtlAutomatonConstSharedPtr ltlAutomaton, PolyhedralAbstractionConstSharedPtr abstraction)
member functions
● points(state: unsigned) : PowersetConstSharedPtr
● productStatePair(state: unsigned) : std::pair<int, int>
● initialStates(): const std::unordered_set<unsigned>&
● acceptingStates(): const std::unordered_set<unsigned>&
● spaceDimension(): PPL::dimension_type
■ buildAutomaton() : void
■ stateDenotationContainsAbstractionPoints(unsigned ltlAutomatonState, unsigned abstractionState) : bool
■ createDummyInitialStateWithEdgesToInitialStates() : void
A
Automaton

Figure 9.9: `PolyhedralSynchronousProductAutomaton` class

- $S' = \{(s, v) \in S \times V \mid pts(v) \subseteq [\![s]\!]\}$: the set of states in the product automaton, where each state is a pair of a state s from the formula automaton and a point v from the polyhedral abstraction, with the condition that the points in v are included in the denotation of state s.
- $\delta' \subseteq S' \times S'$: the set of transitions between the product automaton states.
- $\delta' \subseteq S' \times 2^{\widehat{AP}} \times S'$ is defined by the following rule:

$$\frac{s_1' = (s_1, v_1) \in S' \;\wedge\; s_2' = (s_2, v_2) \in S' \;\wedge\; s_1 \xrightarrow{\alpha}_{\delta} s_2 \;\wedge\; v_1 \longrightarrow_E v_2}{s_1' \xrightarrow{\alpha} s_2'}$$

- $S'_0 = \{(s, v) \in S' \mid s \in S_0\}$: the set of initial states in the product automaton, which correspond to the initial states in the formula automaton.
- $S'_F = \{(s, v) \in S' \mid s \in S_F\}$: the set of accepting states in the product automaton, which correspond to the accepting states in the formula automaton.
- $pts'((s, v)) = pts(v) \subseteq \mathbb{R}^n$: the point set function for the product automaton, which is equal to the point set of v in the polyhedral abstraction.

The `PolyhedralSynchronousProductAutomaton` class implements the synchronous product described above. It combines the functionality of the formula automaton, `PolyhedralLtlAutomaton`, and the polyhedral abstraction, `PolyhedralAbstraction`, to create a product automaton that captures the behaviour of both. The `PolyhedralSynchronousProductAutomaton` class exposes methods for getting the points associated with a state via the method `points(state)`, which returns a `Powerset`, and through the method `acceptance()`, which returns an instance of `AutomatonAcceptance`, allowing us to determine the type of acceptance used by the formula automaton. This information is crucial for determining which emptiness check algorithm to execute next (see Section 9.5).

9.5 Automata Emptiness Check Algorithms

Once the synchronous product between the formula automaton and the polyhedral abstraction is constructed using the `PolyhedralSynchronousProductAutomaton` class (see Section 9.4), emptiness check algorithms can be applied to address the existential denotation problem. The algorithm varies depending on the acceptance type of the formula automaton, whether it is NFA or Büchi. In both cases, the first step is to construct the transposed automaton of `PolyhedralSynchronousProductAutomaton`. After that:

- **NFA**: The algorithm collects all initial states that are reachable from accepting states.
- **Büchi**: The algorithm identifies all non-trivial strongly connected components containing at least one accepting state, and then collects all initial states that are reachable from these components.

Finally, once the initial states are collected, the result is the union of all points associated with the reachable initial states. These algorithms are implemented in the `mc-solvers/emptiness` component, specifically in the header file `emptiness_algorithms.h`. The main function is `collectInitialStatesWithAcceptingRuns`, which accepts a `PolyhedralSynchronousProductAutomaton` as input and returns all reachable initial states based on the acceptance type. This function is used by the `OmnidirectionalSolver` (see Chapter 10) to then compute the union of all points associated with these initial states.

9.6 On-the-Fly Algorithm for Finite-Time Semantics

As discussed in Section 4.4.1, constructing a general polyhedral abstraction is computationally infeasible. However, for finite-time semantics, we can avoid explicitly constructing the full polyhedral abstraction and instead rely solely on the formula automaton, as discussed in Section 4.4.1. The algorithm described in Algorithm 1 is implemented by the `DenotOnTheFly` class, shown in Figure 9.10. The `DenotOnTheFly` class requires a reference to the `PolyhedralSystem` to access the pre-flow (see Section 7.2.4), which is necessary for applying the reachability operators, as well as other details such as the space dimension and the symbol table. Additionally, it needs a reference to an instance of `BackwardNFA`, the transposed NFA of the formula, specifically designed for use by the `DenotOnTheFly` algorithm. The map V, used to ensure termination by tracking the patches associated with the open states, is implemented using a `std::vector<Powerset>` of size equal to the number of states in the automaton. This vector associates, for each open state s, the visited patches of $[\![s]\!]$. Initially, each state is associated with an empty `Powerset`. The class `DenotOnTheFly` implements the operator `()` and returns a `PowersetConstSharedPtr`. The algorithm starts from the accepting states of the `BackwardNFA` and invokes the `denot()` method for each patch of the accepting states. During execution, the `DenotOnTheFly` class collects several statistics, such as the total number of `denot()` calls and the total number of reach operator invocations. If the `m_collectPaths` flag is enabled, it collects all paths contributing to a result in a vector, while also tracking the number

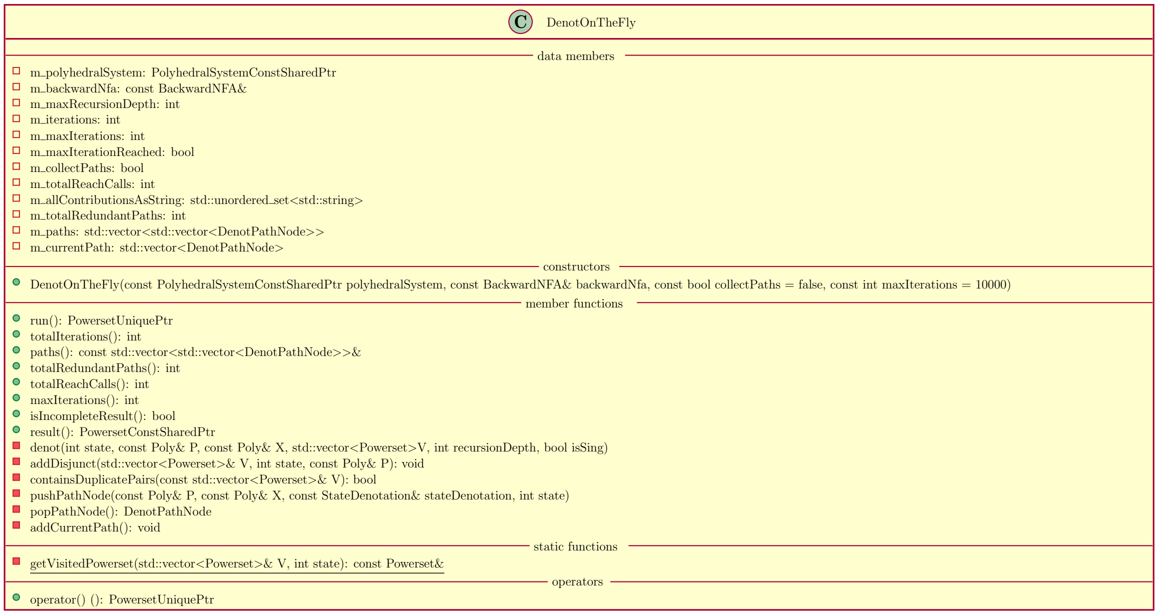

C
DenotOnTheFly
data members
m_polyhedralSystem: PolyhedralSystemConstSharedPtr
m_backwardNfa: const BackwardNFA&
m_maxRecursionDepth: int
m_iterations: int
m_maxIterations: int
m_maxIterationReached: bool
m_collectPaths: bool
m_totalReachCalls: int
m_allContributionsAsString: std::unordered_set<std::string>
m_totalRedundantPaths: int
m_paths: std::vector<std::vector<DenotPathNode>>
m_currentPath: std::vector<DenotPathNode>
constructors
DenotOnTheFly(const PolyhedralSystemConstSharedPtr polyhedralSystem, const BackwardNFA& backwardNfa, const bool collectPaths = false, const int maxIterations = 10000)
member functions
run(): PowersetUniquePtr
totalIterations(): int
paths(): const std::vector<std::vector<DenotPathNode>>&
totalRedundantPaths(): int
totalReachCalls(): int
maxIterations(): int
isIncompleteResult(): bool
result(): PowersetConstSharedPtr
denot(int state, const Poly& P, const Poly& X, std::vector<Powerset>V, int recursionDepth, bool isSing)
addDisjunct(std::vector<Powerset>& V, int state, const Poly& P): void
containsDuplicatePairs(const std::vector<Powerset>& V): bool
pushPathNode(const Poly& P, const Poly& X, const StateDenotation& stateDenotation, int state)
popPathNode(): DenotPathNode
addCurrentPath(): void
static functions
getVisitedPowerset(std::vector<Powerset>& V, int state): const Powerset&
operators
operator() (): PowersetUniquePtr

Figure 9.10: `DenotOnTheFly` class

of redundant paths, i.e. those that produce a contribution already present. Additionally, the `DenotOnTheFly` class allows the specification of a maximum number of `denot()` calls, with a default value of 10,000. This is useful because for huge automata or for state denotations involving two or more patches, the problem can become intractable. Setting this limit allows the algorithm to terminate earlier, yielding an incomplete result. If this occurs, the method `isIncompleteResult()` returns `true`. The `DenotOnTheFly` class is used by the `FiniteOnTheFlySolver`, which is discussed, along with all other solvers, in the next chapter (Chapter 10).

–10–

Solver Hierarchy

CONTENTS: **10.1 Solver Interface. 10.2 Explicit Solvers.** 10.2.1 Omnidirectional Solvers. **10.3 On-The-Fly Solvers. 10.4 Solver Selection Flow Diagrams.**

This chapter presents the hierarchy of solvers implemented in the tool, each designed to compute the existential or universal denotation of a polyhedral system with respect to a given `RTL` formula. The solvers differ in the way they construct and explore the system's behaviour, depending on the chosen semantics (finite, infinite, may, or must), the structural properties of the system (e.g. omnidirectional flow, compactness), and the syntactic fragment of the `RTL` formula. All solvers inherit from a common abstract interface and implement specific algorithms tailored to the scenario at hand, ranging from explicit construction of polyhedral abstractions to on-the-fly computations over automata.

10.1 Solver Interface

All solvers inherit from the abstract `Solver` class, as depicted in Figure 10.1. A `Solver` class requires at least the following components:

- A reference to an instance of `PolyhedralSystem`;
- A reference to the `RTL` formula.

The `TranslationOptimizationFlags` class defines optimisation flags for Spot, which can be passed to `spot::translator` to set the desired optimisation level (see Section 6.3.6). Additionally, the constructor of the `Solver` class takes a boolean flag to specify whether the algorithm should compute the existential or universal denotation. Finally, each `Solver` is assigned a name, with the default being "Solver". The abstract `Solver` class overrides the `()` operator, returning an instance of `SolverResult`. The `SolverResult` class encapsulates the result in the form of a `PowersetSharedPtr` and

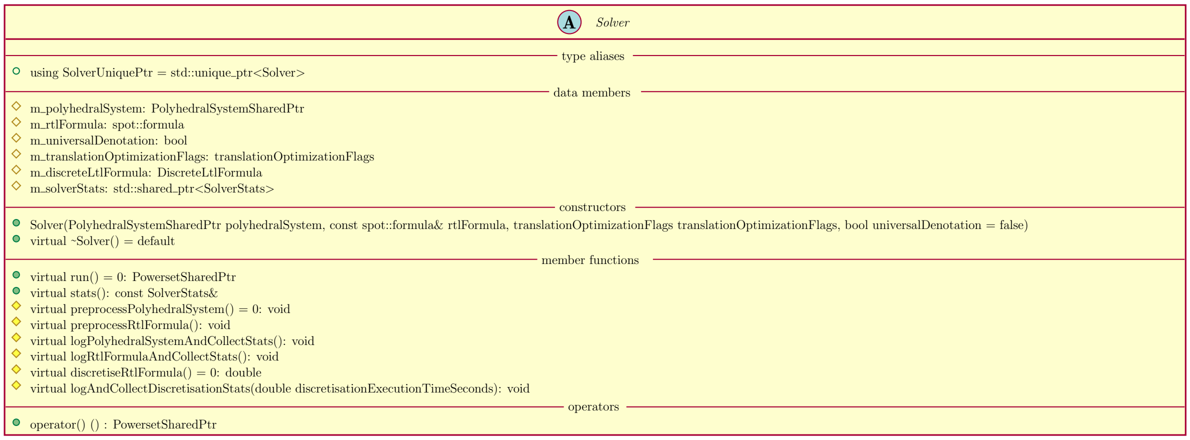

A
Solver
type aliases
using SolverUniquePtr = std::unique_ptr<Solver>
data members
m_polyhedralSystem: PolyhedralSystemSharedPtr
m_rtlFormula: spot::formula
m_universalDenotation: bool
m_translationOptimizationFlags: translationOptimizationFlags
m_discreteLtlFormula: DiscreteLtlFormula
m_solverStats: std::shared_ptr<SolverStats>
constructors
Solver(PolyhedralSystemSharedPtr polyhedralSystem, const spot::formula& rtlFormula, translationOptimizationFlags translationOptimizationFlags, bool universalDenotation = false)
virtual ~Solver() = default
member functions
virtual run() = 0: PowersetSharedPtr
virtual stats(): const SolverStats&
virtual preprocessPolyhedralSystem() = 0: void
virtual preprocessRtlFormula(): void
virtual logPolyhedralSystemAndCollectStats(): void
virtual logRtlFormulaAndCollectStats(): void
virtual discretiseRtlFormula() = 0: double
virtual logAndCollectDiscretisationStats(double discretisationExecutionTimeSeconds): void
operators
operator() () : PowersetSharedPtr

Figure 10.1: `Solver` abstract class

includes a boolean flag indicating whether the result is complete or incomplete (see Section 9.6). The following protected virtual methods are part of the `Solver` class:

- `preprocessPolyhedralSystem()`: This method must be invoked before running the algorithm to solve for the existential or universal denotation. It performs the necessary operations on the polyhedral system before starting the resolution process.

- `preprocessRtlFormula()`: This method must be called before discretising the `RTL` formula. By default, it negates the `RTL` formula if the `m_universalDenotation` flag is set to true.

- `discretiseRtlFormula()`: This method discretises the `RTL` formula, producing an instance of `DiscreteLtlFormula` (see Section 8.2.3).

- Other methods focus on logging operations and saving statistics related to the polyhedral system, the `RTL` formula, and the discretisation process.

10.2 Explicit Solvers

The abstract class `ExplicitSolver` extends `Solver` and must be implemented by solvers that explicitly construct the polyhedral abstraction and solve the existential or universal denotation problem by performing the synchronous product between the polyhedral abstraction and the `RTL` formula automaton. Figure 10.2 shows the abstract class `ExplicitSolver`. The `ExplicitSolver` class introduces several new protected virtual methods:

- `constructPolyhedralAbstraction()`: constructs an instance of `PolyhedralAbstraction` (see Section 9.3).

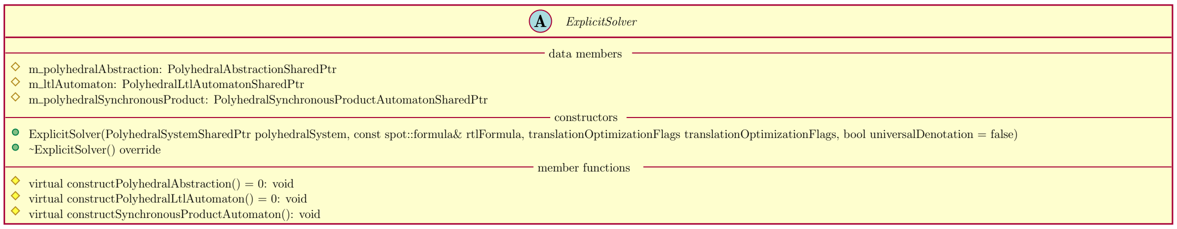

A
ExplicitSolver
data members
m_polyhedralAbstraction: PolyhedralAbstractionSharedPtr
m_ltlAutomaton: PolyhedralLtlAutomatonSharedPtr
m_polyhedralSynchronousProduct: PolyhedralSynchronousProductAutomatonSharedPtr
constructors
ExplicitSolver(PolyhedralSystemSharedPtr polyhedralSystem, const spot::formula& rtlFormula, translationOptimizationFlags translationOptimizationFlags, bool universalDenotation = false)
~ExplicitSolver() override
member functions
virtual constructPolyhedralAbstraction() = 0: void
virtual constructPolyhedralLtlAutomaton() = 0: void
virtual constructSynchronousProductAutomaton(): void

Figure 10.2: `ExplicitSolver` abstract class

- `constructPolyhedralLtlAutomaton()`: constructs an instance of `PolyhedralLtlAutomaton` that represents the discretised `RTL` formula.
- `constructSynchronousProductAutomaton()`: constructs an instance of `PolyhedralSynchronousProductAutomaton` (see Section 9.4) after the polyhedral abstraction and the formula automaton has been constructed.

10.2.1 Omnidirectional Solvers

An extension of the `ExplicitSolver` class is the abstract class `OmnidirectionalSolver`, which overrides the `constructPolyhedralAbstraction()` method to build an `OmnidirectionalPolyhedralAbstraction` (see Section 9.3.1). There are three implementations of the `OmnidirectionalSolver` class:

- `OmnidirectionalFiniteSolver`: Solves the existential or universal denotation problem for an $\texttt{RTL}_\mathsf{f}$ formula.
- `OmnidirectionalInfiniteSolver`: Solves the existential or universal denotation problem for an $\texttt{RTL}_\omega$ formula.
- `OmnidirectionalMaySolver`: Solves the existential or universal denotation problem for an $\texttt{RTL}_\infty$ formula with may semantics.

Figure 10.3 illustrates the hierarchy of omnidirectional solvers. A solver for an $\texttt{RTL}_\infty$ formula with must semantics does not exist because, given the omnidirectional flow, there cannot be finite trajectories that end on the boundary of the invariant and, for each direction, exit the invariant. This is because the trajectory can remain stationary, as $\mathbf{0} \in int(Flow)$.

10.3 On-The-Fly Solvers

The `FiniteOnTheFlySolver` use the on-the-fly algorithm described in Section 4.4.1. After discretising the input formula, this solver constructs the `BackwardNFA` automaton (see Section 9.2.5) and executes the on-the-fly algorithm using the `DenotOnTheFly`

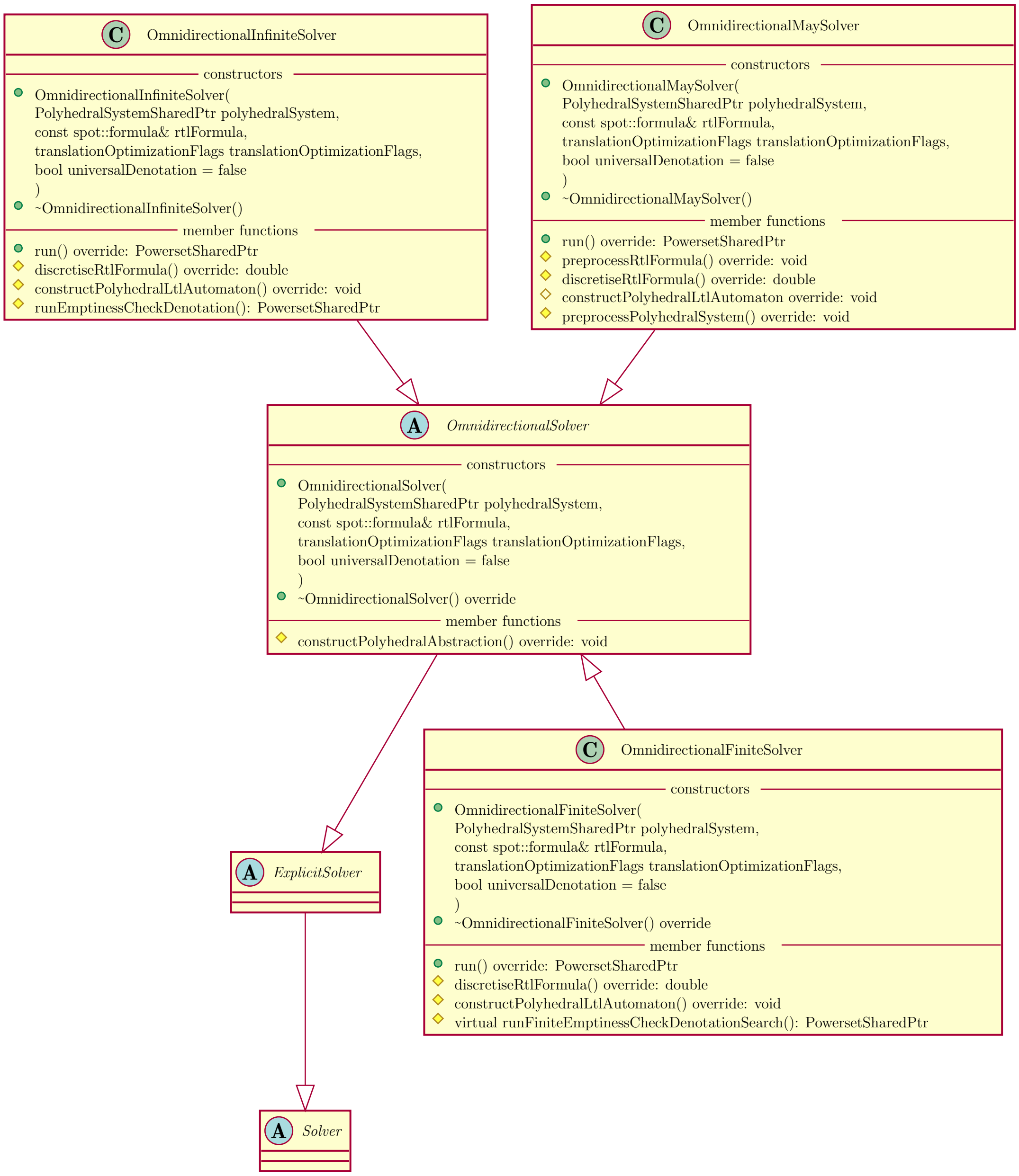

C OmnidirectionalInfiniteSolver
constructors
OmnidirectionalInfiniteSolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false
)
~OmnidirectionalInfiniteSolver()
member functions
run() override: PowersetSharedPtr
discretiseRtlFormula() override: double
constructPolyhedralLtlAutomaton() override: void
runEmptinessCheckDenotation(): PowersetSharedPtr
C OmnidirectionalMaySolver
constructors
OmnidirectionalMaySolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false
)
~OmnidirectionalMaySolver()
member functions
run() override: PowersetSharedPtr
preprocessRtlFormula() override: void
discretiseRtlFormula() override: double
constructPolyhedralLtlAutomaton override: void
preprocessPolyhedralSystem() override: void
A OmnidirectionalSolver
constructors
OmnidirectionalSolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false
)
~OmnidirectionalSolver() override
member functions
constructPolyhedralAbstraction() override: void
A ExplicitSolver
C OmnidirectionalFiniteSolver
constructors
OmnidirectionalFiniteSolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false
)
~OmnidirectionalFiniteSolver() override
member functions
run() override: PowersetSharedPtr
discretiseRtlFormula() override: double
constructPolyhedralLtlAutomaton() override: void
virtual runFiniteEmptinessCheckDenotationSearch(): PowersetSharedPtr
A Solver

Figure 10.3: Hierarchy of omnidirectional solvers

class (see Section 9.6). This is the only solver that may produce an incomplete result. Since large automata and state denotations with patch cardinality greater than or equal to 2 can lead to intractable computations, the number of calls to `denot()` can be bounded to ensure early termination. Figure 10.4 shows the class `FiniteOnTheFlySolver` and its extensions. The following subclasses extend the functionality of `FiniteOnTheFlySolver` to support specific scenarios:

- `StayFiniteOnTheFlySolver`: This class extends the polyhedral system by computing the interpretation of the atomic proposition *stay* and adds it to the system. It then modifies the formula $\mathtt{RTL}_\omega$ by conjoining it with $\mathtt{F}(\mathit{stay} \wedge \mathit{last})$, effectively turning it into a finite-time formula $\mathtt{RTL}_\mathrm{f}$. This solver is used in the scenario involving a non-recurrent $\mathtt{RTL}_\omega$ formula and a compact flow (see Section 4.5.2).

- `BrinkFiniteOnTheFlySolver`: This class computes the interpretation of the atomic proposition `brink` and adds it to the polyhedral system. It modifies the formula $\mathtt{RTL}_\omega$ by conjoining it with $\mathtt{F}(\mathit{brink} \wedge \mathit{last})$. This solver is used when the system has a forced-motion flow and a bounded invariant, and the goal is to solve the existential denotation problem for an $\mathtt{RTL}_\infty$ formula under the may semantics (see Section 4.6.3).

- `BrinkStayFiniteOnTheFlySolver`: This solver combines the techniques of the two solvers above. It is used in scenarios with non-recurrent $\mathtt{RTL}_\infty$ formulae (under either may or must semantics) and compact flow (see Section 4.6.4).

Section 10.4 presents a selection flow diagram that illustrates how the tool chooses the appropriate solver based on the current scenario.

10.4 Solver Selection Flow Diagrams

The `RTL` Model Checking Tool selects the most appropriate solver based on three key elements:

1. The chosen semantics: `fin`, `inf`, `may`, or `must`;
2. The structural properties of the polyhedral system, such as omnidirectional flow, compactness, and forced motion;
3. The syntactic class of the `RTL` formula, in particular whether the formula is non-recurrent (see Section 3.4).

Figures 10.5 and 10.6 illustrate the selection process using decision flow diagrams:

- Figure 10.5 covers the semantics `fin` (finite-time) and `inf` (infinite-time);
- Figure 10.6 covers the semantics `may` and `must`.

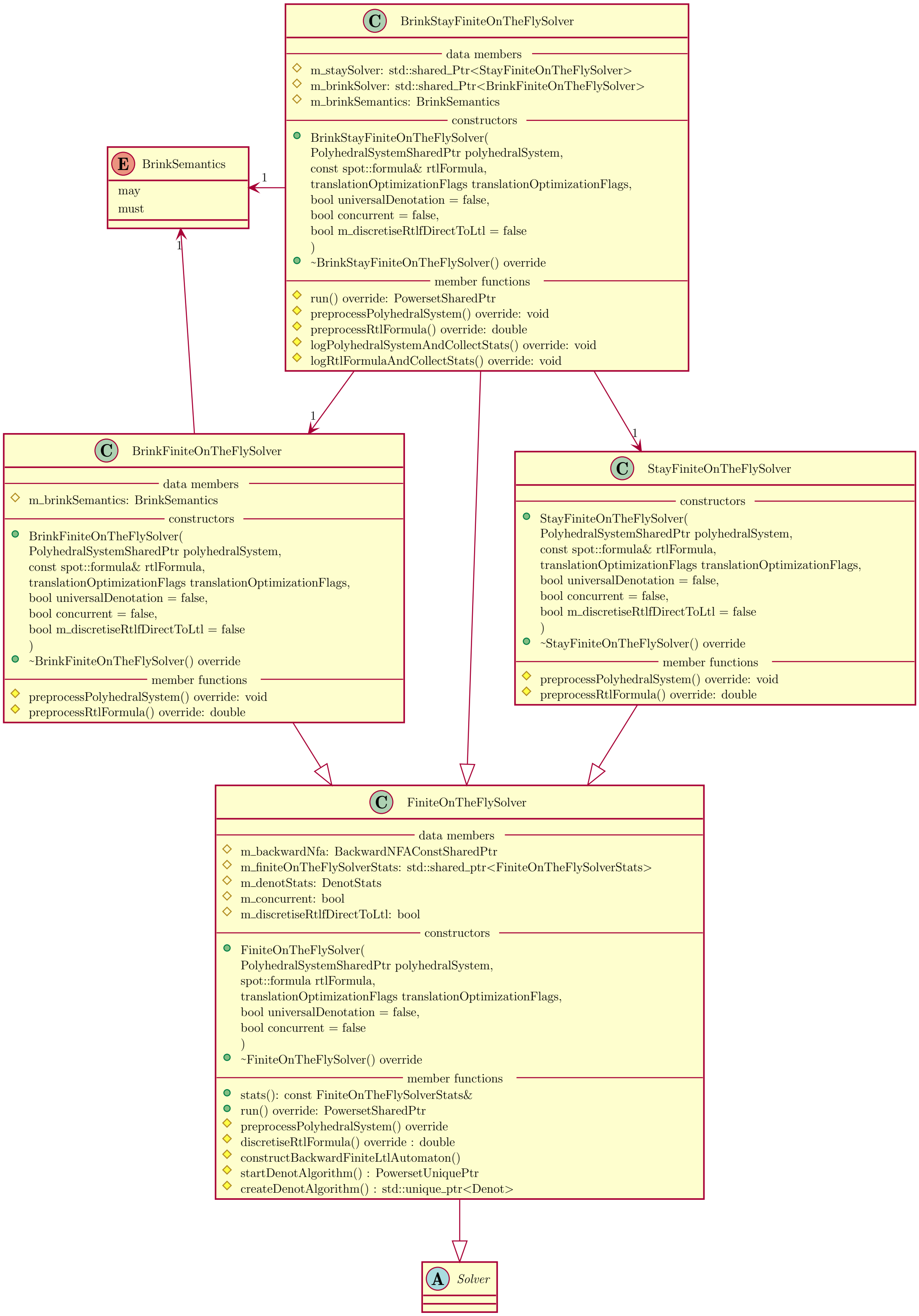

C BrinkStayFiniteOnTheFlySolver
data members
m_staySolver: std::shared_Ptr<StayFiniteOnTheFlySolver>
m_brinkSolver: std::shared_Ptr<BrinkFiniteOnTheFlySolver>
m_brinkSemantics: BrinkSemantics
constructors
BrinkStayFiniteOnTheFlySolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false,
bool concurrent = false,
bool m_discretiseRtlfDirectToLtl = false
)
~BrinkStayFiniteOnTheFlySolver() override
member functions
run() override: PowersetSharedPtr
preprocessPolyhedralSystem() override: void
preprocessRtlFormula() override: double
logPolyhedralSystemAndCollectStats() override: void
logRtlFormulaAndCollectStats() override: void
E BrinkSemantics
may
must
1
1
1
1
C BrinkFiniteOnTheFlySolver
data members
m_brinkSemantics: BrinkSemantics
constructors
BrinkFiniteOnTheFlySolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false,
bool concurrent = false,
bool m_discretiseRtlfDirectToLtl = false
)
~BrinkFiniteOnTheFlySolver() override
member functions
preprocessPolyhedralSystem() override: void
preprocessRtlFormula() override: double
C StayFiniteOnTheFlySolver
constructors
StayFiniteOnTheFlySolver(
PolyhedralSystemSharedPtr polyhedralSystem,
const spot::formula& rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false,
bool concurrent = false,
bool m_discretiseRtlfDirectToLtl = false
)
~StayFiniteOnTheFlySolver() override
member functions
preprocessPolyhedralSystem() override: void
preprocessRtlFormula() override: double
C FiniteOnTheFlySolver
data members
m_backwardNfa: BackwardNFAConstSharedPtr
m_finiteOnTheFlySolverStats: std::shared_ptr<FiniteOnTheFlySolverStats>
m_denotStats: DenotStats
m_concurrent: bool
m_discretiseRtlfDirectToLtl: bool
constructors
FiniteOnTheFlySolver(
PolyhedralSystemSharedPtr polyhedralSystem,
spot::formula rtlFormula,
translationOptimizationFlags translationOptimizationFlags,
bool universalDenotation = false,
bool concurrent = false
)
~FiniteOnTheFlySolver() override
member functions
stats(): const FiniteOnTheFlySolverStats&
run() override: PowersetSharedPtr
preprocessPolyhedralSystem() override
discretiseRtlFormula() override : double
constructBackwardFiniteLtlAutomaton()
startDenotAlgorithm() : PowersetUniquePtr
createDenotAlgorithm() : std::unique_ptr<Denot>
A Solver

Figure 10.4: Class hierarchy of on-the-fly solvers.

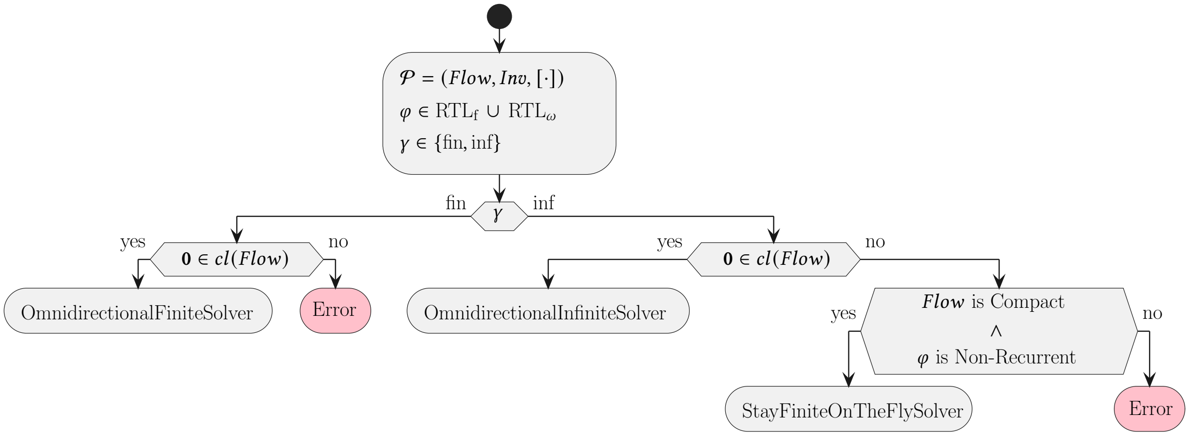

$\mathcal{P} = (Flow, Inv, [\cdot])$
$\varphi \in \mathrm{RTL_f} \cup \mathrm{RTL}_\omega$
$\gamma \in \{\mathrm{fin}, \mathrm{inf}\}$
fin
γ
inf
yes
$\mathbf{0} \in cl(Flow)$
no
OmnidirectionalFiniteSolver
Error
yes
$\mathbf{0} \in cl(Flow)$
no
OmnidirectionalInfiniteSolver
yes
Flow is Compact
∧
φ is Non-Recurrent
no
StayFiniteOnTheFlySolver
Error

Figure 10.5: Solver selection diagram for `fin` and `inf` semantics.

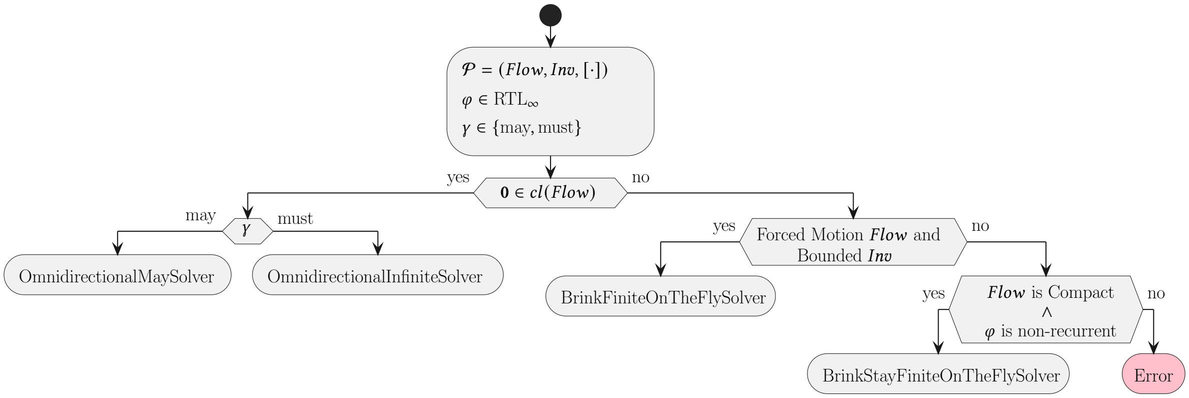

$\mathcal{P} = (Flow, Inv, [\cdot])$
$\varphi \in \mathrm{RTL}_\infty$
$\gamma \in \{\mathrm{may}, \mathrm{must}\}$
yes
$\mathbf{0} \in cl(Flow)$
no
may
γ
must
OmnidirectionalMaySolver
OmnidirectionalInfiniteSolver
yes
Forced Motion Flow and Bounded Inv
no
BrinkFiniteOnTheFlySolver
yes
Flow is Compact
∧
φ is non-recurrent
no
BrinkStayFiniteOnTheFlySolver
Error

Figure 10.6: Solver selection diagram for `may` and `must` semantics.

Each solver receives, in addition to the system and the formula, a boolean flag indicating whether to compute the *universal* or *existential* denotation. While this flag is not shown in the diagrams, it is explicitly passed at runtime and influences the interpretation of the formula and the analysis result. For ***brink***-based solvers (`BrinkFiniteOnTheFlySolver` and `BrinkStayFiniteOnTheFlySolver`), the selected semantics (`may` or `must`) is also passed as a parameter. This choice determines how the interpretation of the auxiliary atomic proposition `brink` is computed, which in turn affects how the system is extended to reduce the `RTL` formula to a finite-time analysis problem. This selection process ensures that the tool chooses the best solver for the given combination of input system, formula structure, and semantics. Unsupported or currently unsolved scenarios are detected at runtime and reported with an informative error message.

Part III

Experimental Framework and Results

–11–

Experimental Infrastructure and Tools

CONTENTS: **11.1 `RTL` Formulae Generation with `rtl-gen`.** **11.2 Polyhedral System Generation with `sys-gen`.** 11.2.1 GAP Experiment: Generalisation and Implementation. **11.3 From Polyhedra to GeoGebra: The `poly-ggb` Tool.** 11.3.1 Usage and Options – 11.3.2 Polyhedra Specification for `poly-ggb` – 11.3.3 Example: Visualising an Instance of the GAP Experiment. **11.4 Built-in Statistics Engine of `rtl-mc`.**

11.1 `RTL` Formulae Generation with `rtl-gen`

The `rtl-gen` tool is designed to automatically generate families of `RTL` formulae for use in testing, benchmarking, and experimental evaluation. It supports a subcommand-based interface, where each subcommand represents a parametric family of formulae. This modular design facilitates future extensions by simply adding new subcommands that generate new classes of formulae.

Command Structure

The general usage of `rtl-gen` is:

```
rtl-gen [--help] [--version] {subcommand}
```

Each `subcommand` corresponds to a family of `RTL` formulae, and may accept its own set of arguments.

Alternation Subcommand

Currently, the only implemented subcommand is `alternation`, which generates formulae that require a trajectory to alternate k times between two atomic propositions p and q. The syntax is as follows:

```
rtl-gen alternation [--help] [--version] [-k VAR] [-p VAR] [-q VAR]
```

- `-k`, `--alternations`: The number of alternations to enforce. Default is 1.
- `-p`: The name of the first atomic proposition. Default is `"p"`.
- `-q`: The name of the second atomic proposition. Default is `"q"`.

Given a value k, the generated formula has the following recursive structure:

$$\mathsf{F}(q_1 \wedge \mathsf{F}(p_1 \wedge \mathsf{F}(q_2 \wedge \ldots)))$$

- Default invocation with one alternation:

  ```
  rtl-gen alternation
  ```

  Output: `F(q & Fp)`

- Four alternations between p and q:

  ```
  rtl-gen alternation -k 4
  ```

  Output: `F(q & F(p & F(q & F(p & Fq))))`

- Same structure but using custom propositions:

  ```
  rtl-gen alternation -k 4 -p "a" -q "b"
  ```

  Output: `F(b & F(a & F(b & F(a & Fb))))`

11.2 Polyhedral System Generation with `sys-gen`

The `sys-gen` tool is responsible for generating instances of polyhedral systems that can be used for model checking against `RTL` specifications. Like `rtl-gen`, the tool is organised around a subcommand-based interface, where each subcommand corresponds to a parametric family of polyhedral systems. The general usage of `sys-gen` is:

```
sys-gen [--help] [--version] {subcommand}
```

- `-h`, `--help`: Displays the help message.
- `-v`, `--version`: Prints version information.
- `subcommand`: Specifies the family of polyhedral systems to generate.

Currently, the only available subcommand is:

- `gap`: Generates a polyhedral system for the *GAP experiment* (see Section 11.2.1), a scalable benchmark involving n interconnected tanks.

This subcommand is discussed in detail in the next subsection (Section 11.2.1). Each invocation of `sys-gen gap` produces a polyhedral system in a format that can be directly parsed and verified by the `RTL` Model Checking Tool (see Chapter 7).

11.2.1 GAP Experiment: Generalisation and Implementation

We introduce the *GAP experiment*, a parametric class of polyhedral systems modelling the interaction of multiple interconnected tanks. We begin by illustrating the case with two tanks, before presenting the generalised version in the following paragraphs.

The 2-Tank Scenario Consider a physical system composed of two tanks, connected by a pump and containing a liquid. The dynamics are governed by the following uncertain, time-varying flow rates:

- An inlet pipe pours liquid into the first tank at a rate in the interval $[1, 2]$;
- A pump transfers liquid from the first tank to the second, with a rate also in $[1, 2]$;
- An outlet extracts liquid from the second tank at a rate in $[0, 3]$.

Let the variables a and b represent the liquid levels in the first and second tanks, respectively. We introduce an additional variable t to model the passage of time. Under these assumptions, the system can be described by the following differential inclusions:

$$\begin{aligned} \dot{a} &\in [-1, 1], \\ \dot{b} &\in [-2, 2], \\ \dot{a} + \dot{b} &\in [-2, 2], \\ \dot{t} &= 1. \end{aligned}$$

These define the Flow component of the polyhedral system. The system is unconstrained in this respect and allows negative values of a and b; however, we impose a polyhedral invariant to enforce non-negativity:

$$\mathit{Inv} \;=\; \{a \geq 0 \;\&\; b \geq 0\}\,.$$

Property Specification Suppose we are interested in computing the set of initial states from which the system can, within the first 10 time units:

1. Reach a configuration where the level in the first tank exceeds that of the second by at least 1 unit;
2. Later, reach a configuration where the second tank exceeds the first by at least 1 unit.

This requirement can be formalised using the `RTL` specification language as follows. We define the following atomic propositions:

$$\begin{aligned}[p] &= \{a \geq b + 1\},\\ [q] &= \{b \geq a + 1\},\\ [t0] &= \{t = 0\},\\ [t1] &= \{t \leq 10\}.\end{aligned}$$

Then, the desired property is captured by the formula:

$$\varphi_{gap} = t0 \ \land \ t1 \ \land \mathsf{G}(Inv) \ \land \ \mathsf{F}\, p \ \land \ \mathsf{F}\, q.$$

Note that the $\mathsf{G}(Inv)$ conjunct can be safely omitted, as the tool implicitly enforces the invariant by intersecting each atomic proposition with the invariant region. This example demonstrates how absolute time constraints (e.g. $t \leq 10$) can be encoded within the system model using additional variables. In Chapter 12, we use this setup to evaluate the effectiveness of various solver strategies for computing the existential denotation of φ_{gap} over the 2-tank polyhedral system.

Command Line Interface The `gap` subcommand supports the following options:

```
Usage: gap [--help] [--version] [--tanks VAR] [--max-time VAR] [--
    thickness VAR]

GAP Experiment with n tanks

Optional arguments:
  -h, --help        shows help message and exits
  -v, --version     prints version information and exits
  -t, --tanks       The total number of tanks [default: 2]
  -m, --max-time    Includes a clock t with derivative 1 and two atomic
                    propositions:
                    t0 { t = 0 }
                    t1 { t <= [max-time] }
                    These propositions define a time window and
                    can be used in RTL formulae to express
```

```
                     temporal constraints. A value of 0 disables
                     this feature (default: 0)
  -g, --thickness  Gap thickness [default: 1]
```

GAP Subcommand - Example with two tanks

Without Time Constraints. The following command generates a 2-tank polyhedral system without time tracking:

```
$ sys-gen gap
```

Output:

```
Inv ( { b >= 0 & a >= 0 } )
Flow { a >= -1 & -a >= -1 & b >= -2 & -b >= -2 & a + b >= -2 & -a - b
   >= -2 }
p ( { a - b >= 1 & b >= 0 } )
q ( { -a + b >= 1 & a >= 0 } )
```

With Time Constraints. The following command includes a clock variable t that progresses at unit rate and defines a temporal window:

```
$ sys-gen gap -m 10
```

Output:

```
Inv ( { b >= 0 & a >= 0 } )
Flow { a >= -1 & -a >= -1 & b >= -2 & -b >= -2 & a + b >= -2 & -a - b
   >= -2 & t = 1 }
p ( { a - b >= 1 & b >= 0 } )
q ( { -a + b >= 1 & a >= 0 } )
t0 ( { t = 0 & b >= 0 & a >= 0 } )
t1 ( { -t >= -10 & b >= 0 & a >= 0 } )
```

When `--max-time` or `-m` is specified, the generated system includes:

- A variable t with constant derivative equal to 1;
- Two atomic propositions:
  - `t0`, representing the condition $t = 0$;
  - `t1`, representing the condition $t \leq$ `[max-time]`.

Generalised GAP Experiment Figure 11.1 illustrates the generalised GAP experiment with n tanks arranged sequentially.

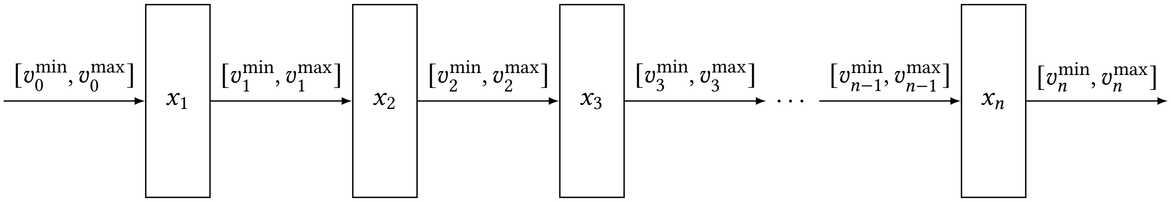

$[v_0^{\min}, v_0^{\max}]$
x_1
$[v_1^{\min}, v_1^{\max}]$
x_2
$[v_2^{\min}, v_2^{\max}]$
x_3
$[v_3^{\min}, v_3^{\max}]$
$\cdots$
$[v_{n-1}^{\min}, v_{n-1}^{\max}]$
x_n
$[v_n^{\min}, v_n^{\max}]$

Figure 11.1: Generalised GAP experiment with n tanks

Each tank $i \in \{1, \ldots, n\}$ is associated with a variable x_i representing its liquid level. The system includes an inlet, $n-1$ internal transfer pumps, and an outlet. Each arrow in the diagram denotes a flow of liquid induced by a pump operating at an uncertain and time-varying rate constrained within a known interval. Specifically:

- The inlet introduces liquid into the first tank x_1 at a rate within $[v_0^{\min}, v_0^{\max}]$;
- Each internal pump transfers liquid from tank x_i to x_{i+1} with rate bounded by $[v_i^{\min}, v_i^{\max}]$, for $1 \leq i < n$;
- The outlet extracts liquid from the last tank x_n at a rate within $[v_n^{\min}, v_n^{\max}]$.

The goal is to construct a polyhedral flow constraint $\mathit{Flow} \subseteq \mathbb{R}^n$ that precisely encodes these dynamics. To construct the polyhedral flow constraint that captures the dynamics of the system, we derive a set of inequalities that describe how each tank's liquid level can evolve over time. For each tank x_i with $i \in \{1, \ldots, n\}$, the change in its level is determined by two pumps:

- The incoming flow from the previous component (either the inlet or a pump from x_{i-1}), operating in the range $[v_{i-1}^{\min}, v_{i-1}^{\max}]$;
- The outgoing flow to the next component (either the next tank x_{i+1} or the outlet), operating in the range $[v_i^{\min}, v_i^{\max}]$.

Therefore, the derivative $\dot{x}_i$ satisfies:

$$\dot{x}_i \in \left[v_{i-1}^{\min} - v_i^{\max},\ v_{i-1}^{\max} - v_i^{\min}\right]$$

In addition to individual constraints on each variable x_i, the flow model must enforce constraints over contiguous subchains of tanks. Let $x_i, x_{i+1}, \ldots, x_k$ be a window of at least two consecutive tanks with $k - i + 1 \geq 2$. For each subchain $x_i, \ldots, x_k$, we impose the following constraint on the sum of their derivatives:

$$\dot{x}_i + \dot{x}_{i+1} + \cdots + \dot{x}_k \in \left[v_{i-1}^{\min} - v_{k+1}^{\max},\ v_{i-1}^{\max} - v_{k+1}^{\min}\right]$$

It is important to note that, when computing the admissible interval for the sum of the derivatives over a subchain of tanks, only the input interval of the first tank in the chain and the output interval of the last tank are considered. In other words, internal transfer rates between tanks within the subchain do not influence the net flow balance. The constraint:

$$\dot{x}_i + \dot{x}_{i+1} + \cdots + \dot{x}_k \in \left[v_{i-1}^{\min} - v_{k+1}^{\max},\ v_{i-1}^{\max} - v_{k+1}^{\min}\right]$$

is fully determined by:

- $\left[v_{i-1}^{\min},\ v_{i-1}^{\max}\right]$: the input pump interval of the first tank x_i;
- $\left[v_{k+1}^{\min},\ v_{k+1}^{\max}\right]$: the output pump interval of the last tank x_k.

To illustrate this, consider a window of three contiguous tanks: x_i, x_{i+1}, x_{i+2}. Let:

- v_{i-1} be the instantaneous flow rate of the pump entering tank x_i,
- v_i be the internal flow rate from x_i to x_{i+1},
- v_{i+1} be the internal flow rate from x_{i+1} to x_{i+2},
- v_{i+2} be the outflow from tank x_{i+2}.

The derivative of each tank level is given by the difference between incoming and outgoing flows:

$$\dot{x}_i = v_{i-1} - v_i, \quad \dot{x}_{i+1} = v_i - v_{i+1}, \quad \dot{x}_{i+2} = v_{i+1} - v_{i+2}$$

Summing these expressions yields:

$$\dot{x}_i + \dot{x}_{i+1} + \dot{x}_{i+2} = (v_{i-1} - v_i) + (v_i - v_{i+1}) + (v_{i+1} - v_{i+2}) = v_{i-1} - v_{i+2}$$

This result confirms that the internal flows v_i and v_{i+1} cancel out, and only the external input and output matter. Therefore, the constraint becomes:

$$\dot{x}_i + \dot{x}_{i+1} + \dot{x}_{i+2} \in \left[v_{i-1}^{\min} - v_{i+2}^{\max},\ v_{i-1}^{\max} - v_{i+2}^{\min}\right]$$

Atomic Propositions p and q for n Tanks In the generalised GAP experiment with n tanks, the atomic propositions p and q are defined based on the balance between the total water levels of tanks at even and odd positions. Specifically:

- The atomic proposition p holds in those states where the sum of the water levels of the tanks at *even* positions (i.e., indices $0, 2, 4, \ldots$) is greater than or equal to the sum of the water levels at *odd* positions (i.e., indices $1, 3, 5, \ldots$) plus one.
- Conversely, q holds in those states where the sum of the levels at odd positions exceeds the sum at even positions by at least one.

GAP Subcommand - Example with four tanks Consider the following invocation of the `sys-gen` tool for the case of $n = 4$ tanks and a time horizon of 20 units:

```
$ sys-gen gap -t 4 -m 20
Inv ( { d >= 0 & c >= 0 & b >= 0 & a >= 0 } )
Flow { a >= -1 & -a >= -1 & b >= -1 & -b >= -1 & c >= -1 & -c >= -1 &
       d >= -2 & -d >= -2 & a + b >= -1 & -a - b >= -1 &
       a + b + c >= -1 & -a - b - c >= -1 &
       a + b + c + d >= -2 & -a - b - c - d >= -2 &
       b + c >= -1 & -b - c >= -1 & b + c + d >= -2 &
       -b - c - d >= -2 & c + d >= -2 & -c - d >= -2 & t = 1 }
p  ( { a - b + c - d >= 1 & d >= 0 & c >= 0 & b >= 0 & a >= 0 } )
q  ( { -a + b - c + d >= 1 & d >= 0 & c >= 0 & b >= 0 & a >= 0 } )
t0 ( { t = 0 & d >= 0 & c >= 0 & b >= 0 & a >= 0 } )
t1 ( { -t >= -20 & d >= 0 & c >= 0 & b >= 0 & a >= 0 } )
```

In this example:

- The tanks are represented by the variables $a = x_0$, $b = x_1$, $c = x_2$, and $d = x_3$.
- The atomic proposition p checks whether the sum $a + c$ (even-indexed tanks) exceeds $b + d$ (odd-indexed tanks) by at least 1, i.e., $a + c \geq b + d + 1$.
- Conversely, q checks whether $b + d \geq a + c + 1$.

11.3 From Polyhedra to GeoGebra: The `poly-ggb` Tool

`poly-ggb` is a command-line tool designed to automatically generate 2D visualisations of polyhedra from a symbolic specification. The specification syntax is identical to the one described in Section 7.1.1, ensuring seamless integration with the rest of the toolchain. This specification is briefly presented in Section 11.3.2. The output is a `.ggb` file, GeoGebra's native format, fully compatible with the GeoGebra software [34]. A `.ggb` file is internally a compressed ZIP archive containing a structured XML representation of graphical elements, such as geometric regions, labelled axes, and coordinate systems. GeoGebra's GUI can open and interpret this format, offering users a rich set of interactive features, including the ability to export the visualisation to high-quality images. The `poly-ggb` tool has proven extremely useful for this project, especially for automatically producing plots of the results generated by the model-checking algorithms under different polyhedral systems, RTL formulae, and semantic interpretations. For example, a script may generate multiple polyhedral systems with the `sys-gen` tool (see Section 11.2), multiple `RTL` formulae with the `rtl-gen` tool (see Section 11.1), invoke `rtl-mc` to compute results (see Appendix A), and finally use `poly-ggb` to produce visual diagrams for each configuration. This approach, together with the `geogebra` tool (see Section 11.3.3), has been used extensively to automatically generate the images shown in Chapter 12.

11.3.1 Usage and Options

Input Options. A polyhedron specification can be provided using one of the following mutually exclusive modes:

- `--polyhedra-file <FILE>`: Reads the specification from a text file.
- `--polyhedra-string <STRING>`: Provides the specification inline as a string.
- `--polyhedra-stdin`: Reads the specification from standard input (default).

Output Options.

- `--output <filename>`: Specifies the name of the output file. If used without `--only-xml`, a `.ggb` file will be created.
- `--only-xml`: If set, only the `geogebra.xml` file will be generated instead of a full `.ggb` archive.

Axes Configuration.

- `-x-axis <var>`: Specifies the variable to be displayed on the horizontal (X) axis. If not provided, the first declared variable in the input specification is used by default.
- `-y-axis <var>`: Specifies the variable to be displayed on the vertical (Y) axis. If not provided, the second declared variable is used by default.

All remaining variables—those not selected for the axes—are automatically fixed to `0`, unless explicitly assigned a value using the `-fixed` option.

Variable Fixing.

- `--fixed [x=1/3, y=-2/3]`: Assigns fixed rational values to selected variables that are not projected onto the 2D plane. This is necessary when the polyhedron has more than two dimensions, enabling a proper projection onto a 2D view. Variables not explicitly fixed will default to 0.

GeoGebra View Specific Configuration. The tool also allows fine-tuned control of the viewport and coordinate system via:

- `--width <int>`, `--height <int>`: Sets the resolution of the Euclidean view. Default is `1980x1080`.
- `--euclidean-view <xZero> <yZero> <xScale> <yScale>`: Configures the GeoGebra coordinate system. Default values ensure a suitable zoom and centering, but can be customised for advanced layout control.

11.3.2 Polyhedra Specification for `poly-ggb`

The input specification consists of a sequence of atomic propositions and their associated polyhedral regions, optionally followed by a colour code in hexadecimal RGB format (e.g. `0xff0000` for red). Each atomic proposition is interpreted as a distinct geometric region to be visualised in the output file.

```
p { a >= b + 2 } 0x008000
q { b >= a + 2 } 0xff0000
r ({
    t = 0 & a >= 0 & -a + b >= -29 &
    b >= 0 & 3*a - 3*b >= -25
}) 0xff0000
```

11.3.3 Example: Visualising an Instance of the GAP Experiment

In this example, we demonstrate how to generate a 2D visualisation of a verification result for the GAP experiment using the `poly-ggb` tool. Specifically, we focus on the polyhedral system described in Section 11.2.1, using the `sys-gen` tool to create the underlying model. We generate a GAP system with two tanks and a maximum time bound of 10 seconds. To specify the temporal property, we use the `rtl-gen` tool (see Section 11.1) to construct an `RTL` formula with four alternations between atomic propositions p and q. The system and formula are then provided as input to the `rtl-mc` model checker (see Appendix A) to compute the *existential denotation* under finite-time semantics (RTL$_{\mathrm{f}}$). The resulting polyhedral output from `rtl-mc` is then passed to `poly-ggb` to generate a GeoGebra `.ggb` file, which can finally be exported as a `.pdf` using the GeoGebra CLI. The entire process can be executed with the following commands:

Code 11.1 Visualising a GAP instance with 2 tanks and finite-time semantics

```
result=$(sys-gen gap -t 2 -m 10
    | rtl-mc --semantics fin -fs "t0 & G(t1) & $(rtl-gen alternation -k 4)")
echo "r $result 0xff0000" | poly-ggb -x "a" -y "b" -O "gap-fin"
geogebra gap-fin.ggb --export="gap-fin.pdf"
```

- `sys-gen gap -t 2 -m 10`: generates a polyhedral system modelling the GAP experiment with 2 tanks and a maximum time horizon of 10 seconds;

- `rtl-gen alternation -k 4`: generates an `RTL` formula that alternates four times between the atomic propositions p and q, resulting in the following nested temporal structure:

$$\mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q)))))$$

- `rtl-mc -semantics fin -fs "..."`: computes the existential denotation of the formula under finite-time semantics. The full formula is constructed by prefixing temporal bounds:

  $$t_0 \wedge \mathsf{G}(t_1) \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q)))))$$

  The result is `(  t = 0 & b >= 0 & a >= 0 & -b + a >= -23 & b - a >= -21  )`.

- `poly-ggb`: renders the resulting polyhedron in two dimensions by projecting it onto selected axes. In this example, the variable `a` is assigned to the horizontal axis and `b` to the vertical axis, specified via the flags `-x "a"` and `-y "b"`. The polyhedron is labelled as `r` and coloured red using the hexadecimal colour code `0xff0000`. The command produces a GeoGebra file named `gap-fin.ggb`, which can be opened directly in the GeoGebra interface or exported to other formats.

- `geogebra`: invokes the GeoGebra command-line interface to export the resulting `.ggb` file into a `.pdf`.

GeoGebra can also export a `.ggb` file to a `.tikz` image. Figure 11.2 shows the resulting plot obtained from the `gap-fin.ggb` file produced in this example. The red region represents the existential denotation computed by `rtl-mc` under finite-time semantics. When creating a `.ggb` file with `poly-ggb`, the default view settings of GeoGebra may not always produce an image that clearly displays the desired region of the 2D space. This is particularly true when the relevant portion of the polyhedron lies outside the default coordinate window. In such cases, it is necessary to manually adjust the Euclidean view using the `--euclidean-view` option, which sets the origin and scaling of the view via four parameters `(xZero, yZero, xScale, yScale)`. To determine appropriate values for these parameters, one practical approach is to open the generated `.ggb` file in the GeoGebra GUI, visually adjust the view to frame the region of interest, and then inspect the resulting parameters from the interface. These values can then be reused in headless rendering to ensure consistent and meaningful exports across experiments.

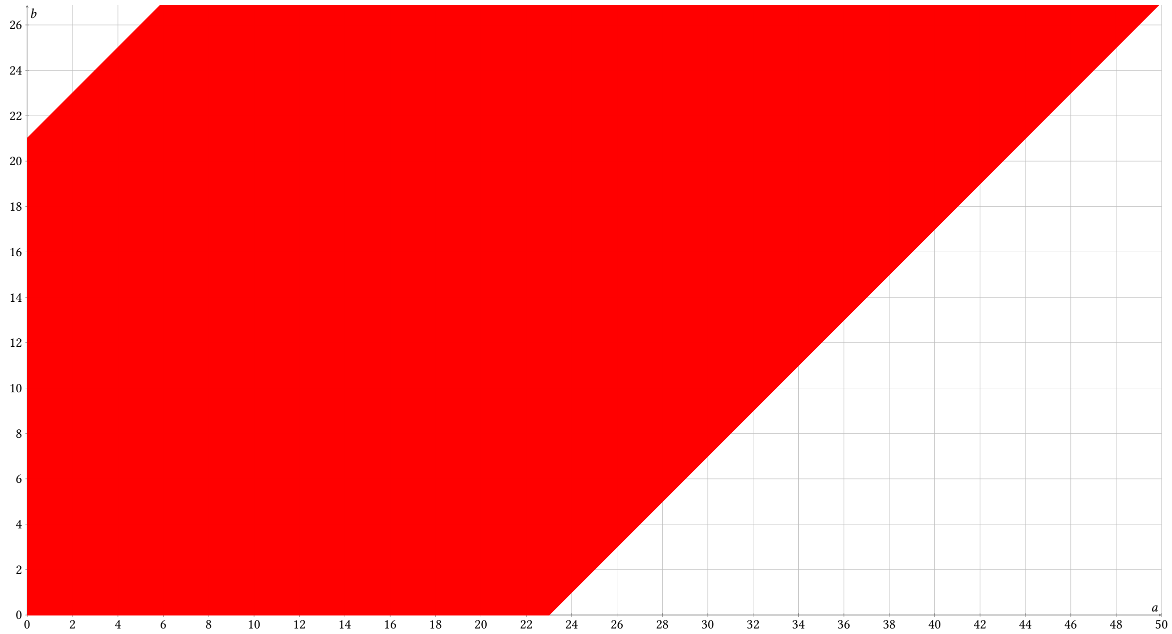


Figure 11.2: Visualisation of the existential denotation for the GAP experiment with 2 tanks and a time bound of 10. The red region corresponds to the set of initial configurations from which the system satisfies the finite-time formula
$t_0 \wedge \mathsf{G}(t_1) \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q)))))$.
The computed result is the polyhedron: `( { t = 0 & b ≥ 0 & a ≥ 0 & -b + a ≥ -23 & b - a ≥ -21 } )`.
The clock variable t is fixed to 0 and eliminated by projection in the final visualisation.

11.4 Built-in Statistics Engine of `rtl-mc`

The `rtl-mc` tool provides a built-in statistics engine that can be enabled via the `--stats` (or `-s`) command-line option (see Appendix A). This engine allows users to retrieve detailed execution metrics about the model-checking process, including information on formula size, polyhedral system characteristics, automaton generation and translation, discretisation, and denotation computation. Each placeholder used with `--stats` acts like an indexed array element and must be provided with an explicit index starting from `0` (e.g. `%As[0]`, `%ATs[1]`, etc.). This is necessary because multiple entities of the same type (e.g. automata, polyhedral systems, solvers) may be generated during verification. The example below runs the model checker on a GAP system with two tanks (see Section 11.2.1) and maximum time bound of 10 seconds, using a temporal formula with four alternations between p and q. It collects the total time spent in the solver and prints it using the `%x[0]` placeholder:

Code 11.2 Collecting solver execution time using the statistics engine

```
sys-gen gap -t 2 -m 10 \
  | rtl-mc --semantics fin \
         -fs "t0 & G(t1) & $(rtl-gen alternation -k 4)" \
         --stats "Total solver execution time: %x[0] seconds."
```

Example output:

```
Total solver execution time: 0.027833 seconds.
```

All available placeholders supported are listed in Tables 11.1, 11.2, and 11.3.

Placeholder	**Description**
— *General Solver Statistics* —	
`%x[i]`	Execution time (in seconds) for solver `i`.
— *RTL Formula Statistics* —	
`%rta[i]`	Number of atomic propositions in RTL formula `i`.
`%rl[i]`	Length (in terms of operators and atoms) of RTL formula `i`.
— *Polyhedral System Statistics* —	
`%Pta[i]`	Number of atomic propositions in polyhedral system `i`.
`%Psp[i]`	Space dimension (number of variables) in polyhedral system `i`.
— *Discretisation Statistics* —	
`%da[i]`	Number of atomic propositions in the discretised RTL formula `i`.
`%dl[i]`	Length of the discretised RTL formula `i`.
`%dx[i]`	Time (in seconds) spent on discretising RTL formula `i`.

Table 11.1: Statistics for RTL formulae, polyhedral systems, and discretisation.

Placeholder	**Description**
— *General Automaton Statistics* —	
`%As[i]`	Number of states in automaton `i`.
`%Ais[i]`	Number of initial states in automaton `i`.
`%Aas[i]`	Number of accepting states in automaton `i`.
`%Ae[i]`	Number of edges in automaton `i`.
`%Ax[i]`	Execution time (in seconds) for constructing automaton `i`.
`%Ascc[i]`	Number of strongly connected components (SCCs) in automaton `i`.
— *Polyhedral LTL Automaton Translation* —	
`%ATx[i]`	Translation time (in seconds) for automaton `i`.
`%ATs[i]`	Number of states in translated automaton `i`.
`%ATis[i]`	Initial states in translated automaton `i`.
`%ATe[i]`	Number of edges in translated automaton `i`.
`%ATase[i]`	Number of accepting sets in translated automaton `i`.
`%ATscc[i]`	Number of SCCs in translated automaton `i`.
— *Optimised Automaton Statistics* —	
`%AOx[i]`	Time (in seconds) to optimise automaton `i`.
`%AOs[i]`	Number of states after optimisation in automaton `i`.
`%AOis[i]`	Initial states after optimisation in automaton `i`.
`%AOas[i]`	Accepting states after optimisation in automaton `i`.
`%AOe[i]`	Edges after optimisation in automaton `i`.
`%AOscc[i]`	SCCs after optimisation in automaton `i`.
— *Patch Statistics* —	
`%Amp[i]`	Maximum number of patches in any state of automaton `i`.
`%Atp[i]`	Total number of patches `i`.

Table 11.2: Statistics related to automaton construction, translation, optimisation, and patches.

Placeholder	**Description**
— Finite Automaton (NFA) Statistics —	
`%ANx[i]`	Execution time (in seconds) to generate NFA `i`.
`%ANs[i]`	Total number of states in NFA `i`.
`%ANis[i]`	Number of initial states in NFA `i`.
`%ANas[i]`	Number of accepting states in NFA `i`.
`%ANe[i]`	Number of edges in NFA `i`.
`%ANscc[i]`	Number of SCCs in NFA `i`.
`%ANr[i]`	Maximum recursive depth (on-the-fly algorithm) in NFA `i`.
— On-the-Fly Denotation Statistics —	
`%Di[i]`	Total number of `denot()` calls for `DenotOnTheFly` `i`.
`%Dx[i]`	Total execution time (in seconds) for `DenotOnTheFly` `i`.
`%Dp[i]`	Total number of paths for `DenotOnTheFly` `i`.
`%Dpr[i]`	Total number of redundant paths (paths with the same contribution) for `DenotOnTheFly` `i`.
`%Drc[i]`	Total reachability computations performed in `DenotOnTheFly` `i`.
`%Dr[i]`	Result of `DenotOnTheFly` `i`.
`%Dd[i]`	`true` if the result is incomplete due to reaching iteration bound (for `DenotOnTheFly` `i`).
`%Dmi[i]`	Maximum number of `denot()` calls allowed (can be set with `DENOT_MAX` environment variable).

Table 11.3: Statistics for NFA and on-the-fly algorithm.

–12–

Model Checking Results and Performance Evaluation

CONTENTS: **12.1 GAP Experiment `RTL` Model Checking Results.** 12.1.1 Incremental GAP Thickness under Different Flow Constraints – 12.1.2 Incremental Time – 12.1.3 Visual Certificates of Existential Denotation Membership. **12.2 Scalability Experiments.** **12.3 `RTL` Automaton Visualisation.** **12.4 Profiling On-the-Fly Execution: Fine-Grained Statistics.**

This chapter presents the experimental evaluation of the `RTL` Model Checking Tool, focusing on both the correctness of the verification results and the performance of the underlying algorithms. The goal is twofold: to demonstrate that the tool produces meaningful and accurate results over a variety of system configurations and semantic settings, and to analyse its scalability and runtime behaviour under increasingly complex scenarios. We leverage the *GAP experiment* (see Section 11.2.1) to systematically vary the flow constraints, spatial gap, and time bounds, to analyse how the existential denotation evolves and to highlight semantic differences across finite-time, may, and must interpretations. We then shift our focus to performance evaluation. Scalability is assessed with respect to both the dimensionality of the system and the logical complexity of the input formula. The chapter also includes visual representations of the generated automata and trajectories. Finally, we exploit the built-in statistics engine to collect fine-grained profiling data, offering insights into performance bottlenecks and algorithmic behaviour.

12.1 GAP Experiment `RTL` Model Checking Results

In this section, we address the existential denotation problem on the polyhedral system described in Section 11.2.1, using two type of `RTL` formulae. The atomic propositions involved in these formulae are interpreted over a state space with two continuous variables, a and b, and one clock variable t. The global invariant of the system is Inv $\{a \geq 0 \land b \geq 0\}$. Table 12.1 presents the interpretations of the atomic propositions alongside the two `RTL` formulae used in the GAP experiment.

Proposition	**Interpretation**
t_0	$\{t = 0\}$
t_1	$\{t \leq 10\}$
p	$\{a \geq b + \delta\}$
q	$\{b \geq a + \delta\}$

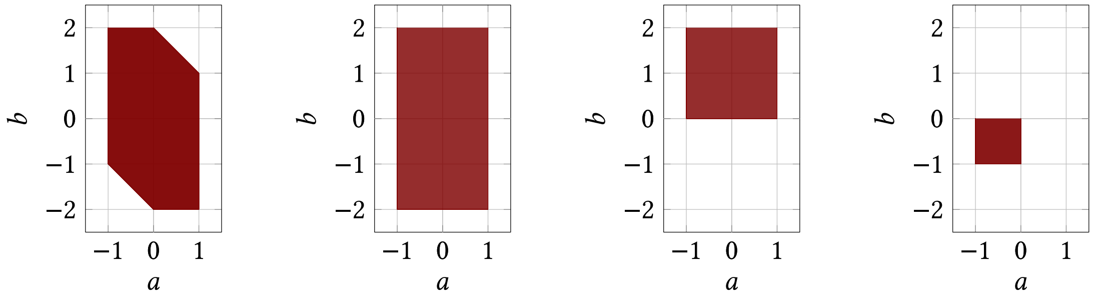
(a) Interpretation of atomic propositions. The constant $\delta > 0$ represents the gap between the regions defined by p and q.

Formula	**Description**
$\varphi_{\text{gap}}^{p,q}$	$t_0 \land \mathsf{G}(t_1) \land \mathsf{F}(p \land \mathsf{F}(q))$
$\varphi_{\text{gap}}^{q,p}$	$t_0 \land \mathsf{G}(t_1) \land \mathsf{F}(q \land \mathsf{F}(p))$

(b) GAP model checking formulae.

Table 12.1: Atomic propositions and formulae used in the GAP model checking experiments.

We consider different types of $Flow$ constraints that define the admissible derivatives for the variables a, b, and the clock t. In all experiments, the clock variable t is present and has a constant derivative equal to 1, unless explicitly stated otherwise, as in the omnidirectional flow experiment, where the clock is omitted. For visualisation purposes, flow diagrams in Figure 12.1 display only the dimensions a and b.



$Flow^{\text{std}}$ $Flow^{\text{rect}}$ $Flow^{\text{up}}$ $Flow^{\text{must}}$

Figure 12.1: Visual representations of the flow constraints used in the GAP experiments.

In the following sections, we show the results obtained by solving the existential denotation problem over the polyhedral system, formulae, and flow constraints introduced above. Most experiments incrementally increase the threshold δ that separates the atomic propositions p and q, while one experiment varies the maximum time bound in the interpretation of t_1, increasing it step by step. All results are visualised through a 2D projection by eliminating the clock variable t from the output (if present). In each plot, the blue region represents the atomic proposition p, the red region corresponds to q, and the yellow region denotes the existential denotation computed by the model checker.

12.1.1 Incremental GAP Thickness under Different Flow Constraints

In this section, we incrementally increase the gap δ between the regions defined by the atomic propositions p and q, increasing it by one unit at each step. For each value of δ, we compute the existential denotation for both $\varphi^{p,q}_{\text{gap}=\delta}$ and $\varphi^{q,p}_{\text{gap}=\delta}$ formulae under different semantics (finite-time, may, and must). Each result is visualised in two dimensions by projecting the computed denotation onto the (a, b)-plane. The plots should be read from left to right, where each successive image corresponds to an incremented value of δ. The following experiments are reported:

- **Finite-time semantics with $Flow^{\text{std}}$:**
  - $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ (Figure 12.2): in this experiment, we analyse how the existential denotation evolves as the gap δ between the atomic regions defined by p and q increases. The yellow area represents the set of initial states from which a trajectory exists that first reaches region p and subsequently region q within the time bound of 10 seconds. As δ increases, the required trajectory must travel a longer distance between p and q, which significantly reduces the set of admissible starting points. Eventually, for $\delta = 15$, the time bound becomes not enough to reach region q from region p, even when the trajectory starts directly within region p. As a result, the existential denotation progressively shrinks and collapses onto region p.
  - $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ (Figure 12.3): this experiment is structurally equivalent to the previous one, except that the formula now requires reaching region q first, followed by region p. The evolution of the existential denotation follows a similar trend, but the yellow area now progressively converges toward region q. A notable difference emerges starting from $\delta = 11$: the entire boundary of region q is no longer included in the existential denotation. For instance, consider a trajectory starting at point $(a = 0, b = 15)$ when $\delta = 11$, where the goal is to reach region p. To do so, the value of a must increase while b must decrease. However, under $Flow^{\text{std}}$, the maximum rate of increase allowed for $\dot{a}$ is 1. Given the time bound of 10 seconds, the trajectory can reach at most $a = 10$, which is not enough to reach region p. No matter the rate of change for $\dot{b}$, the trajectory remains too far from p after 10 seconds. Consequently, starting from $\delta = 11$, the existential denotation no longer fully covers the boundary of region q, due to the limited dynamics permitted by $Flow^{\text{std}}$.
- **May semantics with $Flow^{\text{std}}$:**
  - $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ (Figure 12.4): this experiment illustrates the set of initial states from which there exists a trajectory satisfying the formula $\varphi^{p,q}_{\text{gap}=\delta}$ under the may semantics. That is, the trajectory must reach region p, then region q, and finally terminate at a *may-exit* point (see Section 4.6). In this specific example, any point on the boundary of the invariant qualifies as a may-exit, since the flow constraint $Flow^{\text{std}}$ allows both variables a and b to move in any direction. We observe that for $\delta = 1$ through $\delta = 5$, the existential denotation consists of two overlapping patches. This phenomenon is analysed in more detail in Section 12.1.3, where we explicitly construct trajectories starting from specific initial points to explain this outcome. Starting from $\delta = 6$, the denotation collapses into a single patch, which progressively shrinks as δ increases, until it vanishes completely at $\delta = 10$. The most effective way to satisfy the formula $\varphi^{p,q}_{\text{gap}=\delta}$ is to move along the diagonal orthogonal to both regions p and q. This geometric intuition provides a simple justification for the observed behaviour.

  - $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ (Figure 12.5): this experiment mirrors the previous one, but the formula now requires reaching region q first, followed by region p, and then a may-exit point. For all $0 < \delta < 15$, the existential denotation consists of a single patch that becomes progressively smaller as δ increases, eventually vanishing at $\delta = 15$. Consider, for instance, the point $(a = 5, b = 20)$. Starting from this point, the best way to satisfy $\varphi^{q,p}_{\text{gap}=\delta}$ (for $0 < \delta < 15$) is to follow a trajectory with derivatives $\dot{a} = 1$ and $\dot{b} = -2$. Since the trajectory starts within region q, its goal is to reach p and subsequently a may-exit. Evolving the trajectory with those derivatives allows not only to reach p within the time bound, but also to touch the invariant boundary, satisfying the may-exit requirement.

- **Finite-time semantics with $Flow^{\text{rect}}$:**
  - $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ (Figure 12.6): in this experiment, we switch to the $Flow^{\text{rect}}$ flow constraint, which removes the coupling condition $\dot{a} + \dot{b} \in [-2, 2]$, thus making the derivatives $\dot{a}$ and $\dot{b}$ independent. The results obtained for the formula $\varphi^{p,q}_{\text{gap}=\delta}$ under $Flow^{\text{rect}}$ are identical to those obtained under $Flow^{\text{std}}$ (see Figure 12.2). This outcome can be explained by observing that the most effective way to satisfy the formula is to move along the diagonal orthogonal to both regions p and q at maximum speed. Since both $Flow^{\text{std}}$ and $Flow^{\text{rect}}$ allow such a diagonal motion with the same intensity, the denotation does not change.
  - $\varphi^{q,p}_{\text{gap}=\delta}$ for $\delta = 1$ to $\delta = 15$ (Figure 12.7): In this case, the formula requires reaching q before p. The outcome closely resembles that in Figure 12.3. The only notable difference is that, under the $Flow^{\text{rect}}$ constraint, the existential denotation tends to include more points and exhibits a more rectangular shape.

- **May semantics with $Flow^{\text{rect}}$:**
  - $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ (Figure 12.8): the outcome of this experiment differs from the one shown in Figure 12.4 in that the resulting patches now exhibit a rectangular shape. Starting from $\delta = 6$, the patch takes the form of a square, unlike the more rounded shapes seen previously. This change is due to the use of the $Flow^{\text{rect}}$ constraint, which allows the derivatives $\dot{a}$ and $\dot{b}$ to evolve independently. As a result, the system can now simultaneously increase b and decrease a at maximum intensity, a capability not permitted under $Flow^{\text{std}}$.
  - $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ (Figure 12.9): the result obtained in this experiment is identical to that shown in Figure 12.5 with the $Flow^{\text{std}}$ constraint. This indicates that, for this specific formula and under the may semantics, allowing independent evolution of $\dot{a}$ and $\dot{b}$ under $Flow^{\text{rect}}$ does not provide any additional benefit compared to $Flow^{\text{std}}$.

- **Finite-time semantics with $Flow^{\text{up}}$:**
  - $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ (Figure 12.10): in this experiment, the system evolves under the $Flow^{\text{up}}$ constraint, which only allows the variables a and b to remain constant or increase over time. The existential denotation computed under this constraint is a strict subset of the one obtained with $Flow^{\text{std}}$ in Figure 12.2, since the formula remains unchanged and the inclusion $Flow^{\text{up}} \subset Flow^{\text{std}}$ holds. To satisfy the formula $\varphi^{p,q}_{\text{gap}=\delta}$, the trajectory must first reach region p and then region q. Under $Flow^{\text{up}}$, the only way to do this is by increasing the value of b with the maximal allowed derivative, i.e., $\dot{b} = 2$.

  - $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ (Figure 12.11): this experiment requires trajectories to reach region q first and then region p. To achieve this under $Flow^{\text{up}}$, the variable a must increase with its maximum allowed derivative $\dot{a} = 1$. Compared to the previous case with formula $\varphi^{p,q}_{\text{gap}=\delta}$, the denotation becomes empty earlier. This is because reaching region p from q is constrained by the limited rate of increase of a, while in the previous experiment the transition from p to q was possible due to the higher maximum derivative allowed for b, namely $\dot{b} = 2$.

- **Omnidirectional flow ($Flow^{\text{std}}$ without the clock variable $\dot{t} = 1$):**

  - $\varphi^{p,q}_{\text{gap}=1}$ and $\varphi^{q,p}_{\text{gap}=1}$ under all semantics (Figure 12.12): in this experiment, we deliberately remove the clock variable t, whose derivative is normally fixed to 1, to simulate an omnidirectional flow (see Section 4.4.2). With the clock eliminated, the variable t remains constantly equal to 0, and thus there are no time constraints in the system. Consequently, the existential denotation exactly coincides with the invariant Inv.

- **Must semantics with $Flow^{\text{must}}$:**

  - $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ (Figure 12.13): in this experiment, we use the flow constraint $Flow^{\text{must}}$, which restricts the derivatives $\dot{a}$ and $\dot{b}$ to values in the interval $[-1, 0]$. This flow has been specifically designed for the must semantics. Under must semantics, to satisfy the formula $\varphi^{p,q}_{\text{gap}=\delta}$, the trajectory must first reach region p, then region q, and finally terminate at a *must-exit* point (see Section 4.6). In this configuration, the only valid must-exit point is the origin $(a = 0, b = 0)$. The resulting existential denotation takes the shape of a square that becomes progressively smaller as δ increases. Within each square, it is always possible to reach region q by following a direction where $\dot{a} = -1$ and $\dot{b} = 0$.

  - $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ (Figure 12.14): this experiment requires reaching region q first, followed by region p, and finally the must-exit point at $(a = 0, b = 0)$. The result is symmetric to the previous one. In this case, from any point within the resulting square-shaped existential denotation, it is always possible to reach region p by following a direction in which $\dot{a} = 0$ and $\dot{b} = -1$, which is allowed under the $Flow^{\text{must}}$ constraint.

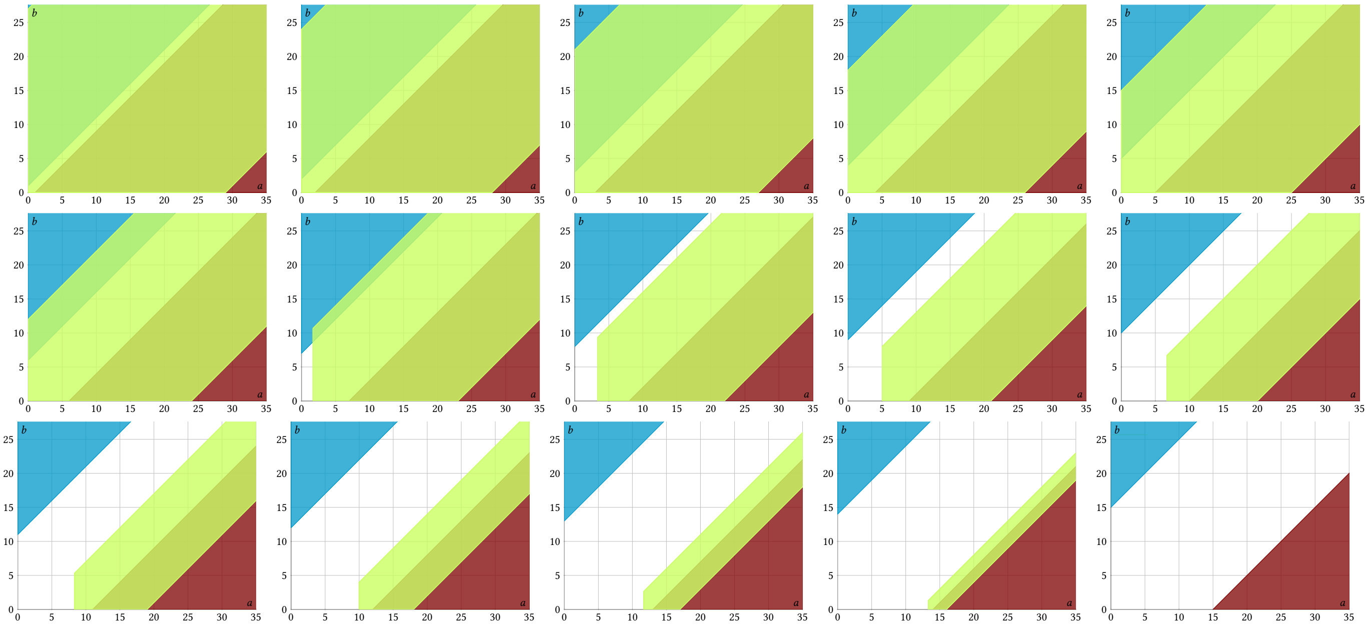


Figure 12.2: Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $\mathit{Flow}^{\text{std}}$.

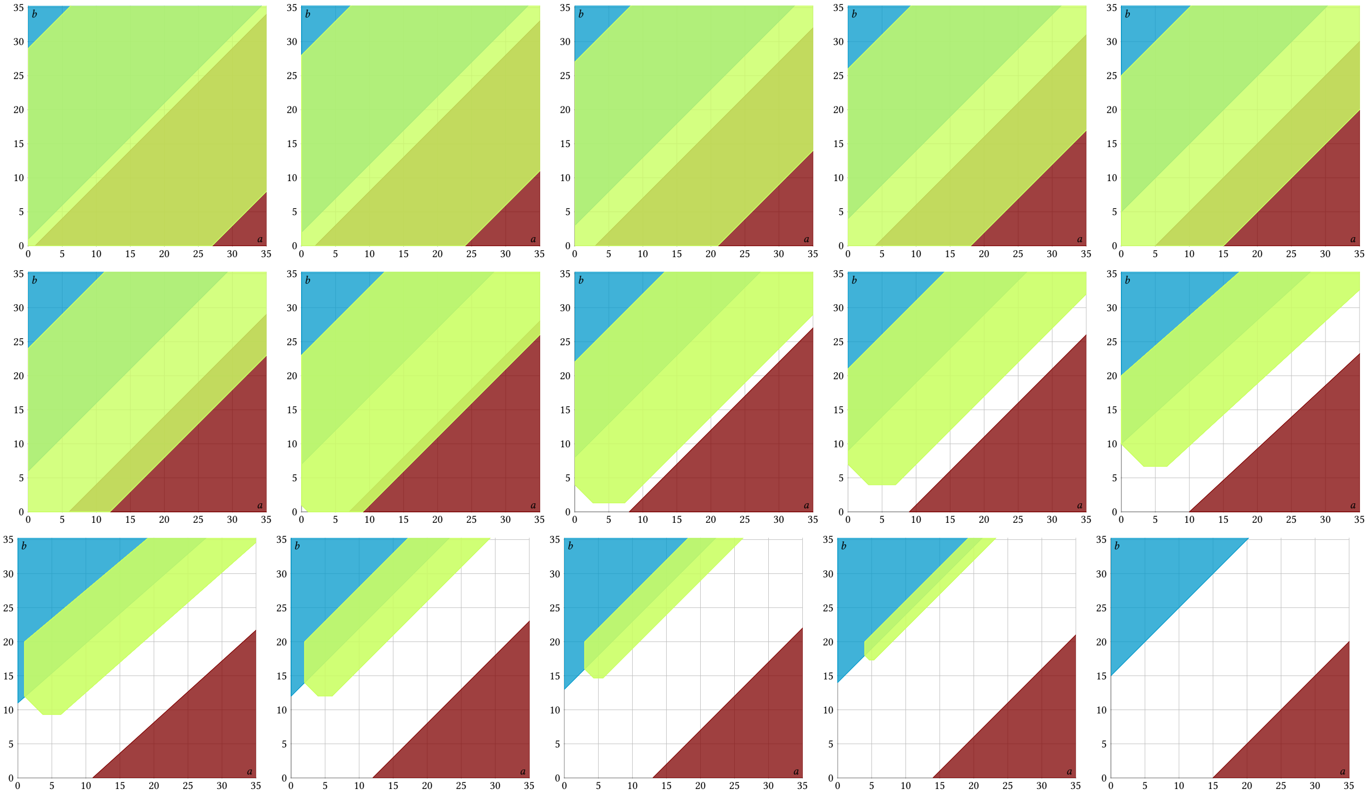


Figure 12.3: Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $\mathit{Flow}^{\text{std}}$.

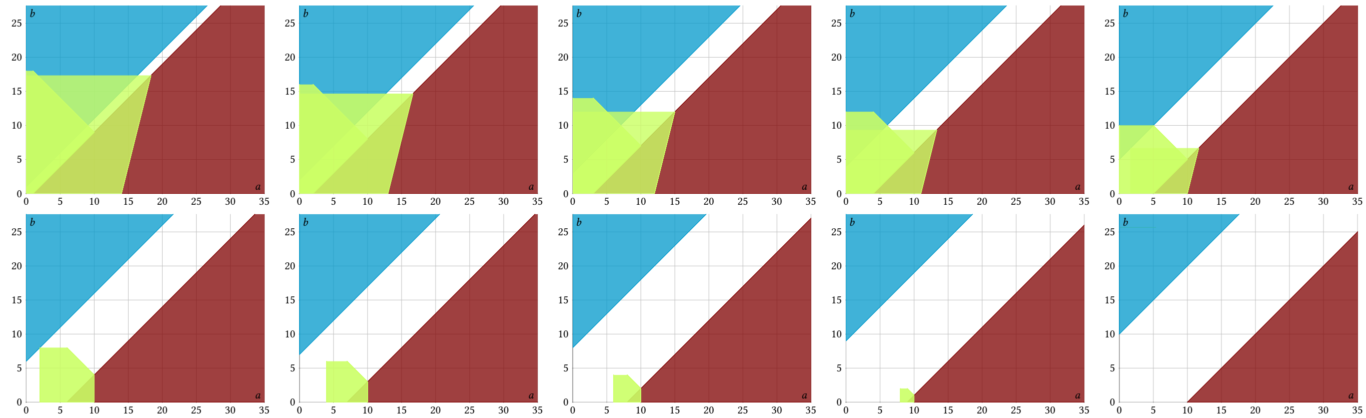


Figure 12.4: Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under may semantics and $\mathit{Flow}^{\text{std}}$.

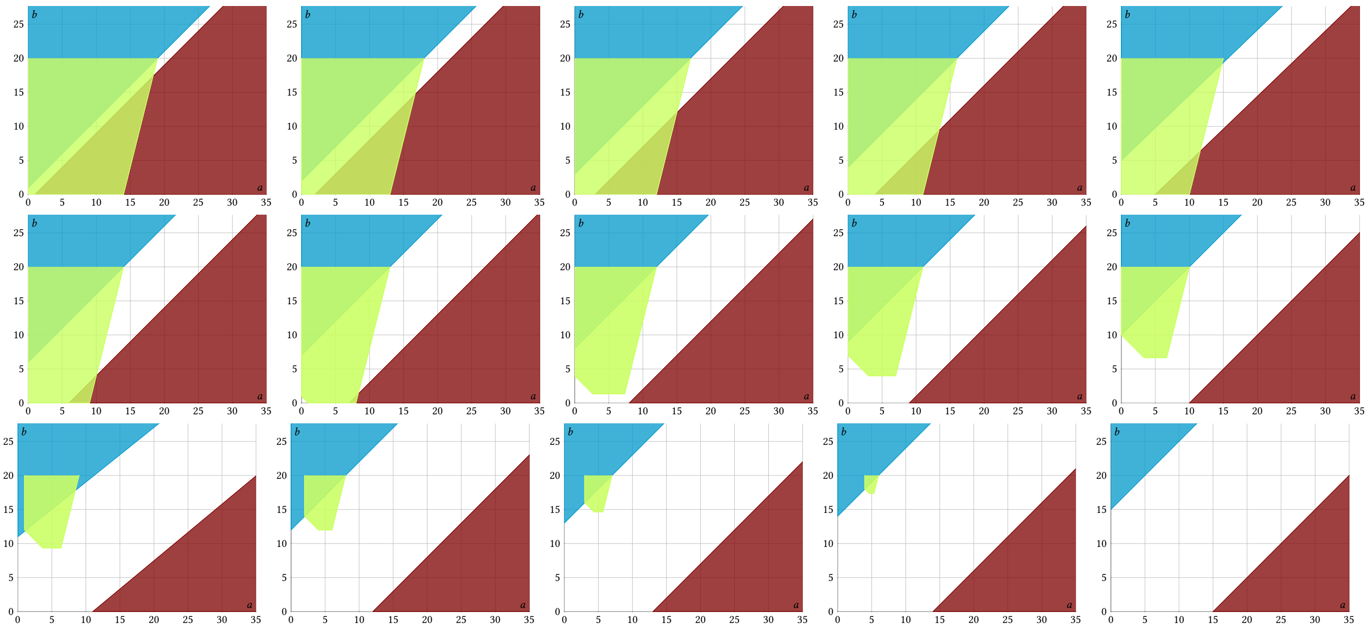


Figure 12.5: Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under may semantics and $\mathit{Flow}^{\text{std}}$.

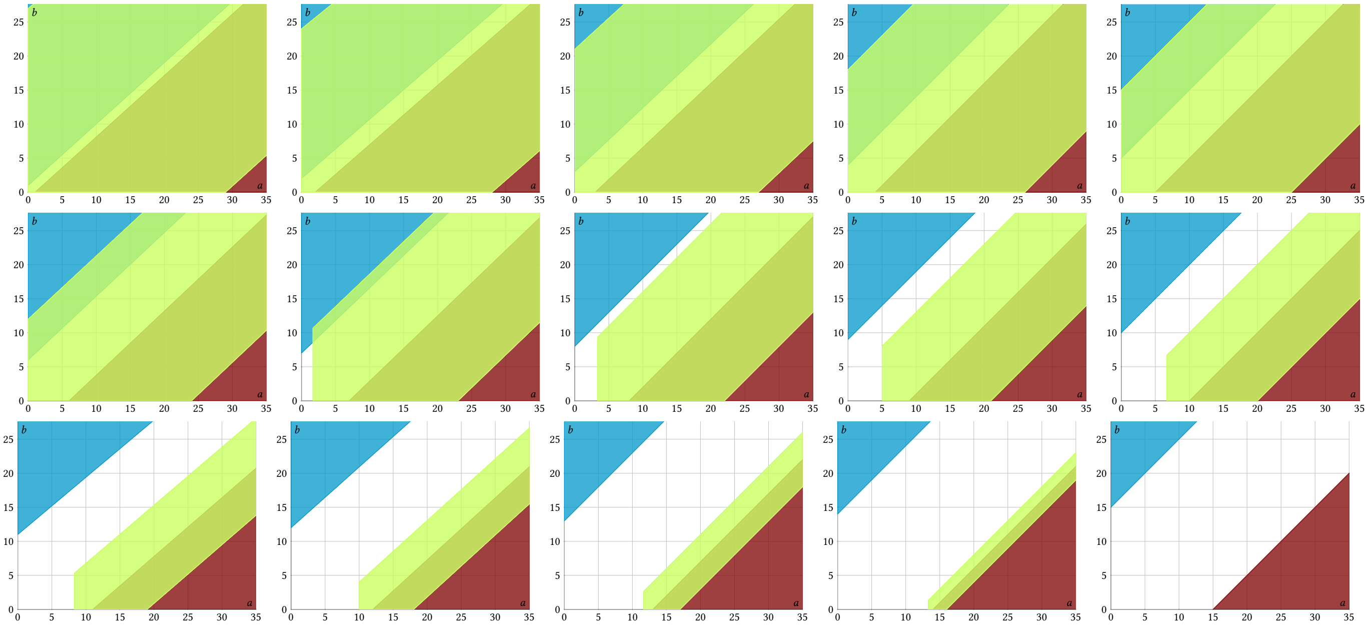


Figure 12.6: Formula $\varphi^{p,q}_{\mathrm{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $\mathit{Flow}^{\mathrm{rect}}$.

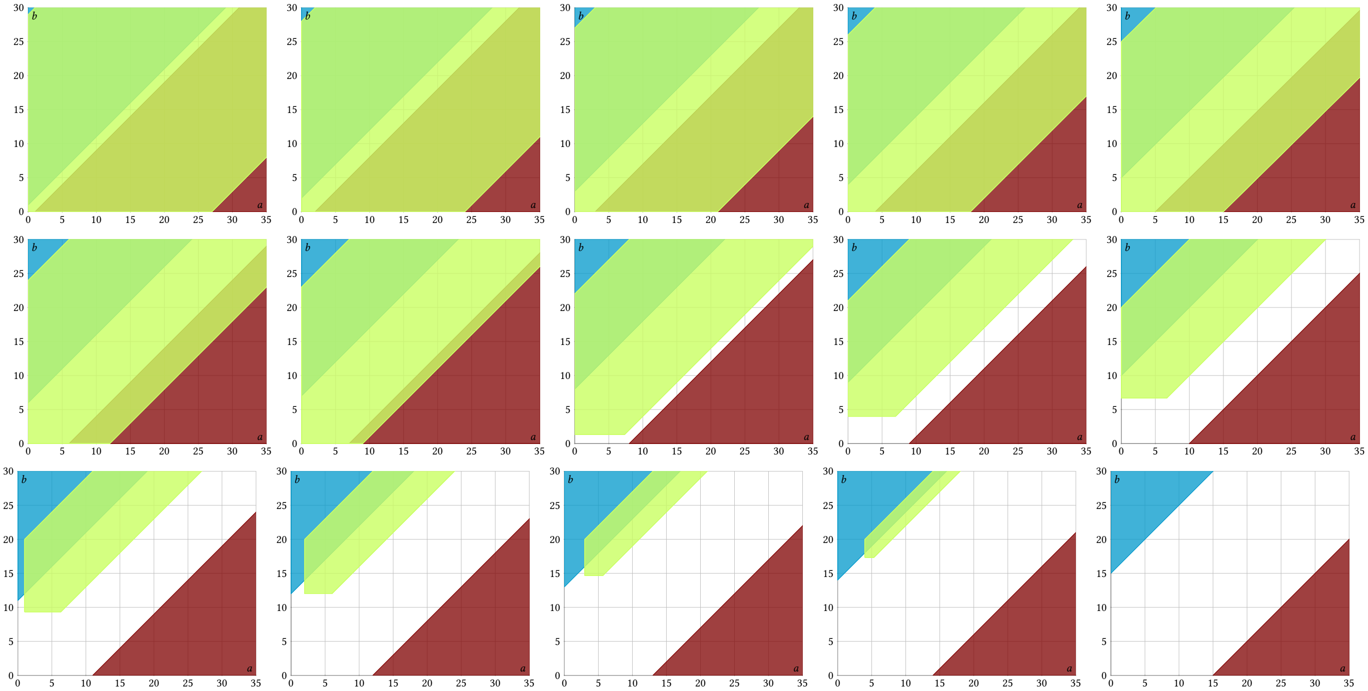


Figure 12.7: Formula $\varphi^{q,p}_{\mathrm{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $\mathit{Flow}^{\mathrm{rect}}$.

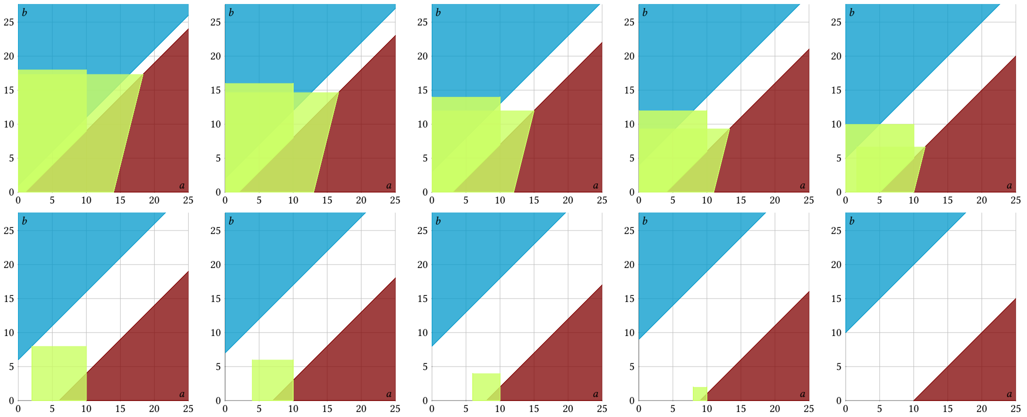


Figure 12.8: Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ under may semantics and $\mathit{Flow}^{\text{rect}}$.

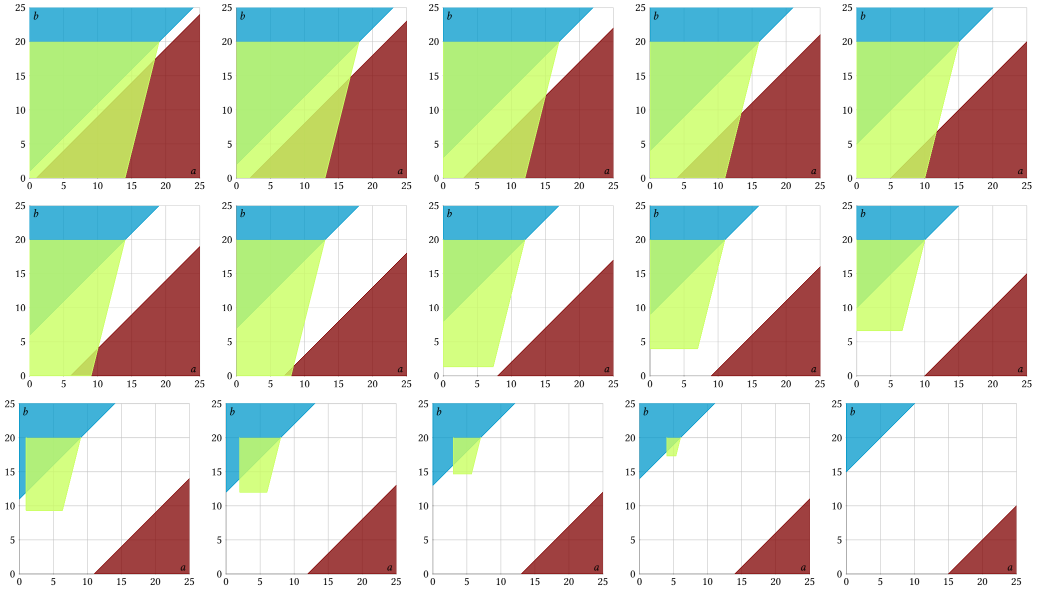


Figure 12.9: Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under may semantics and $\mathit{Flow}^{\text{rect}}$.

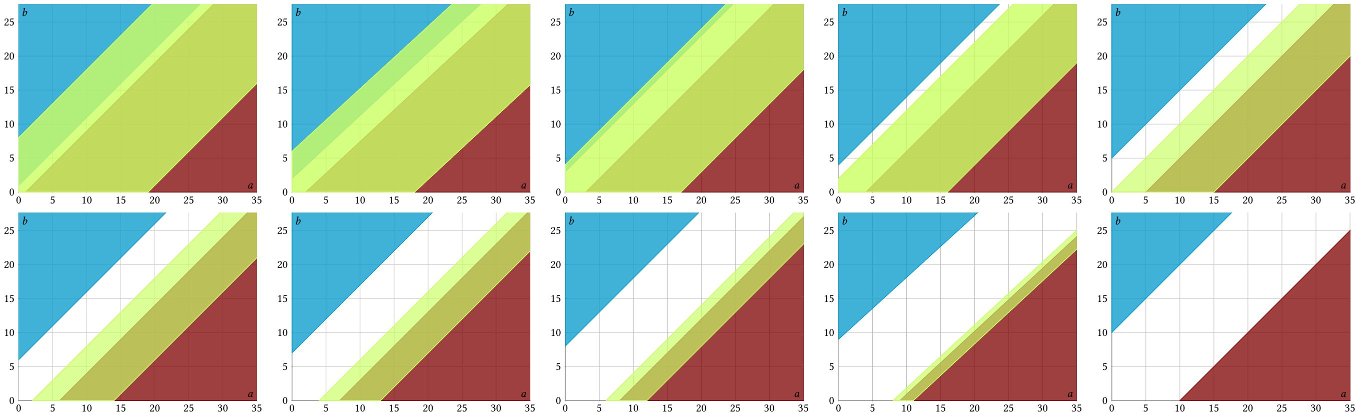


Figure 12.10: Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ under finite-time semantics and $\mathit{Flow}^{\text{up}}$.

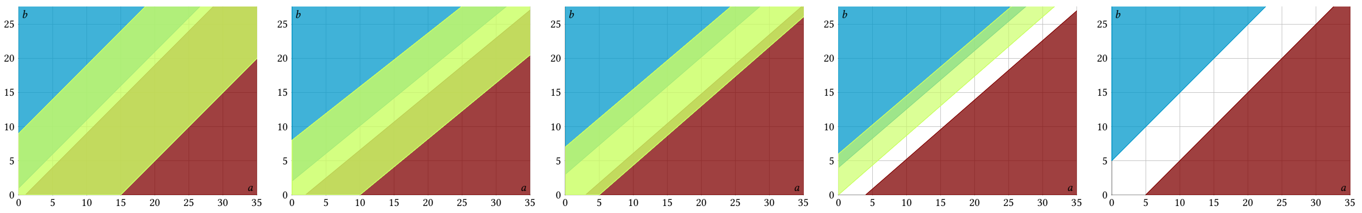


Figure 12.11: Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ under finite-time semantics and $\mathit{Flow}^{\text{up}}$.

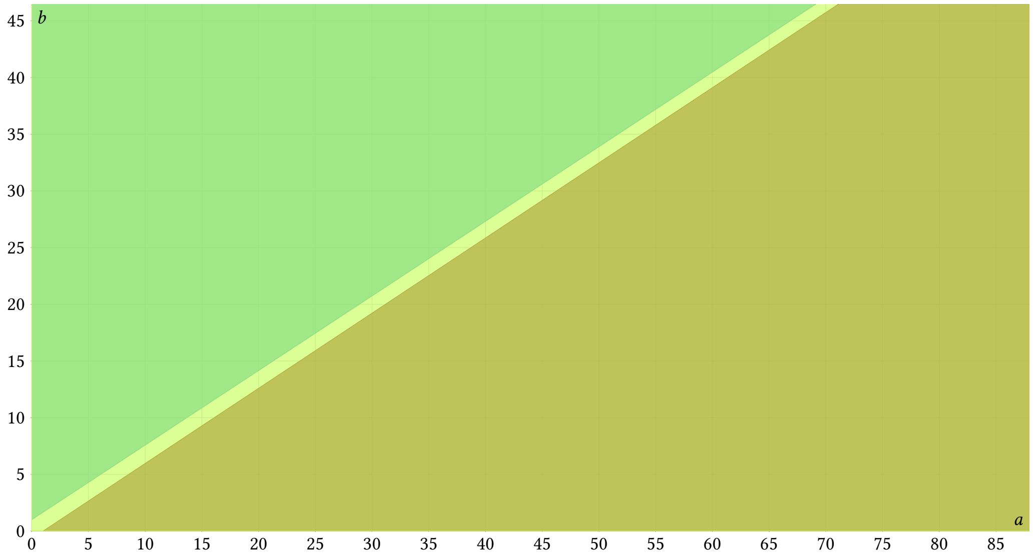


Figure 12.12: $\varphi^{p,q}_{\text{gap}=1}$ and $\varphi^{q,p}_{\text{gap}=1}$ of δ under all semantics and $\mathit{Flow}^{\text{std}}$ without clock variable (omnidirectional flow).

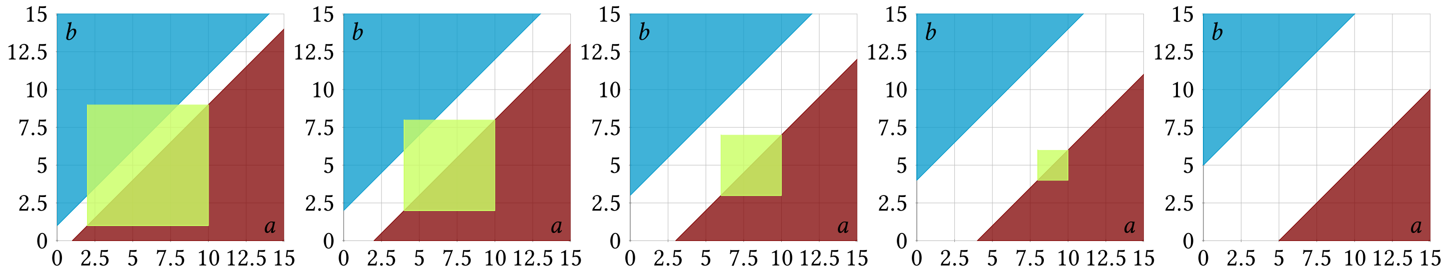


Figure 12.13: Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ under may semantics and $\mathit{Flow}^{\text{must}}$.

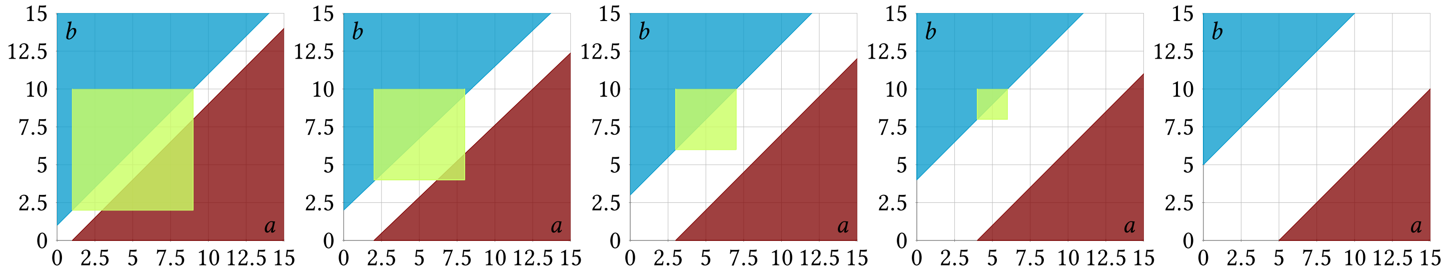


Figure 12.14: Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ under may semantics and $\mathit{Flow}^{\text{must}}$.

12.1.2 Incremental Time

Figure 12.15 presents the results obtained for the formula $\varphi^{p,q}_{\text{gap}=1}$ under finite-time semantics, using the flow constraint $Flow^{\text{std}}$. In this experiment, we gradually increase the maximum time bound t from 1 to 10. As expected, the larger the time allowance, the greater the number of initial states from which a trajectory exists that satisfies the formula. That is, as t increases, the existential denotation expands to include more points that can reach region p and subsequently region q within the time limit.

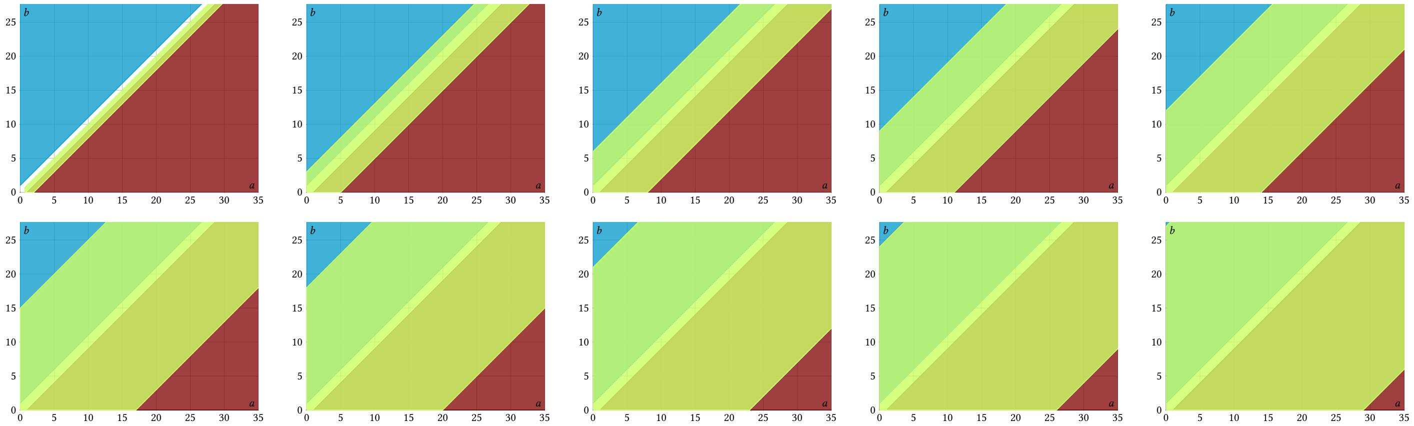


Figure 12.15: Formula $\varphi^{p,q}_{\text{gap}=1}$ with max time from $t = 1$ to $t = 10$ under finite-time semantics and $Flow^{\text{std}}$.

12.1.3 Visual Certificates of Existential Denotation Membership

In this section, we revisit the result obtained for the formula $\varphi^{p,q}_{\text{gap}=1}$ under the *may* semantics with flow constraint $\mathit{Flow}^{\text{rect}}$. Our goal is to provide trajectory-based evidence that specific initial points belong to the existential denotation previously computed. To this end, we select a few representative points and explicitly trace the trajectories that justify their inclusion in the existential denotation. The selected initial points are:

- $A_0 = (10, 18)$
- $B_0 = (0, 18)$
- $C_0 = (11, 17.33)$
- $D_0 = (18.33, 17.33)$

We remind that the maximum available time is 10 seconds and that the clock variable has a constant derivative $\dot{t} = 1$. The flow constraint $\mathit{Flow}^{\text{rect}}$ imposes independent bounds on each derivative: $\dot{a} \in [-1, 1]$ and $\dot{b} \in [-2, 2]$. Figure 12.16 overlays the selected initial points onto the existential denotation and illustrates the corresponding admissible trajectories. Each trajectory reaches region p, then region q, and finally reach the boundary of the invariant, that, in this specific case, is the set of may-exit points.

- **Point A**:

$$A_0 = (10, 18) \xrightarrow[9\text{ s}]{\dot{a}=-1,\ \dot{b}=-2} A_1 = (1, 0) \xrightarrow[1\text{ s}]{\dot{a}=-1,\ \dot{b}=2} A_2 = (0, 2)$$

- **Point B**:

$$B_0 = (0, 18) \xrightarrow[8\text{ s}]{\dot{a}=0,\ \dot{b}=-2} B_1 = (0, 2) \xrightarrow[1\text{ s}]{\dot{a}=1,\ \dot{b}=-2} B_2 = (1, 0) \xrightarrow[1\text{ s}]{\dot{a}=-1,\ \dot{b}=1} B_3 = (0, 1)$$

- **Point C**:

$$\begin{aligned} C_0 = (11, 17.33) &\xrightarrow[1\text{ s}]{\dot{a}=-1,\ \dot{b}=-2} C_1 = (10, 15.33) \\ &\xrightarrow[\approx 3.17\text{ s}]{\dot{a}=0,\ \dot{b}=-2} C_2 = (10, 9) \\ &\xrightarrow[\approx 0.67\text{ s}]{\dot{a}=-1,\ \dot{b}=2} C_3 = (9.33, 10.33) \\ &\xrightarrow[\approx 5.17\text{ s}]{\dot{a}=0,\ \dot{b}=-2} C_4 = (9.33, 0) \end{aligned}$$

- **Point D**:

$$D_0 = (18.33, 17.33) \xrightarrow[\approx 0.67s]{\dot{a}=-1,\ \dot{b}=2} D_1 = (17.66, 18.66) \xrightarrow[\approx 9.33s]{\dot{a}=0,\ \dot{b}=-2} D_2 = (17.66, 0)$$

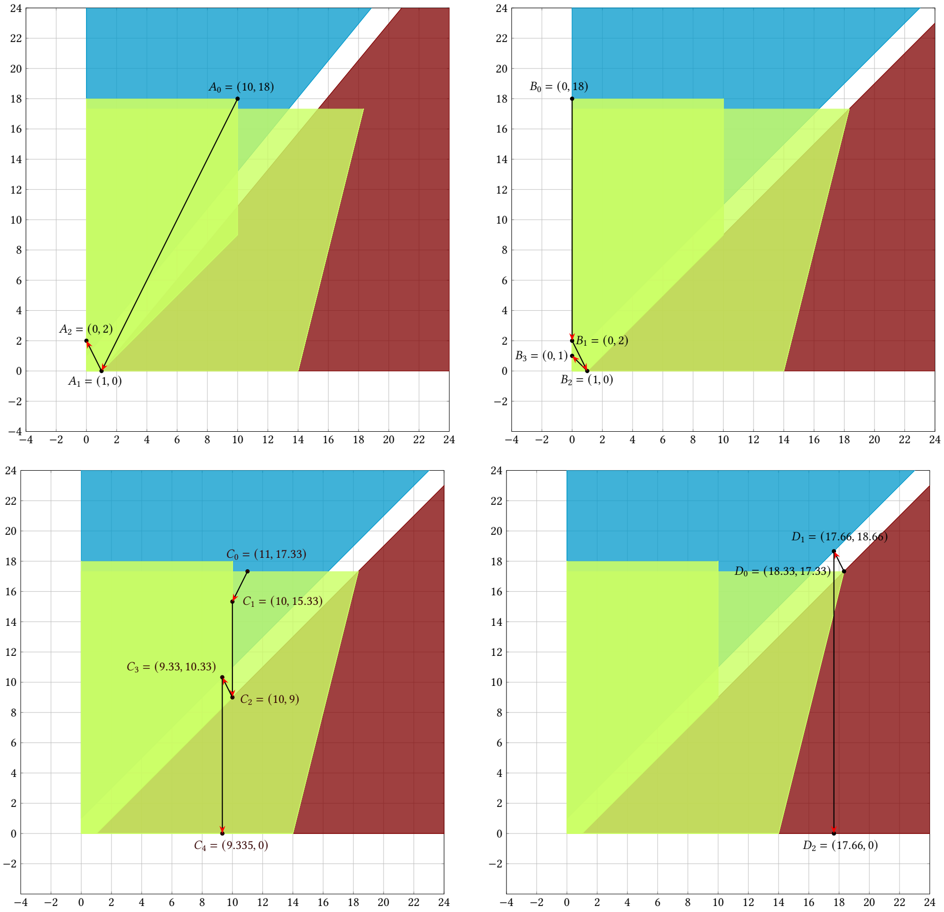


Figure 12.16: Admissible trajectories from selected initial points under may-semantics, formula $\varphi^{p,q}_{\text{gap}=1}$ and $\mathit{Flow}^{\text{rect}}$

12.2 Scalability Experiments

This section investigates the scalability of the tool with respect to the number of variables and the alternation depth of the input formula. To this end, we use the `sys-gen` utility to generate a polyhedral system modelling the GAP experiment (see Section 11.2.1) with a parametric number of tanks and a clock variable t with constant derivative $\dot{t} = 1$. Each system consists of n variables in total, where $n = m + 1$, with m tanks and one global clock. The `rtl-gen` utility is used to generate `RTL` formulae $\varphi_k^{p,q}$ with an arbitrary number of alternations k between the atomic propositions p and q, assuming a fixed spatial gap of 1 between p and q. All experiments were executed on a workstation equipped with an Intel® Core™ i9-13900KF processor (24 cores, 32 threads) and 64 GB of RAM, running a 64-bit Linux system. The experiment is structured as follows:

- For each configuration, we compute the existential denotation of the formula $\varphi_k^{p,q}$ under finite-time semantics. For example, with $k = 1$, we have $t0 \wedge \mathsf{G}(t1) \wedge \mathsf{F}(p \wedge \mathsf{F}(q))$.
- We begin with a system of $n = 3$ variables (i.e. two tanks and one clock) and increase the alternation depth k from 1 to 10.
- After reaching $k = 10$, we increment the number of variables n by adding one tank at a time, repeating the experiment for each new configuration of n while again varying k from 1 to 10.
- Execution time is measured for each configuration to assess scalability.

Table 12.2 reports the total execution time (in seconds) required to solve the existential denotation problem for each combination of alternation depth k and number of variables n. Additional statistics such as the number of automaton states and the total number of calls to the $reach$ operators (see Section 2.5) are reported in Table 12.3. These values remain constant regardless of the number of system variables n, as they are solely determined by the structure of the formula. These statistics refer to the `BackwardNFA` automaton (see Section 9.2.5) constructed specifically for use with the on-the-fly solver `FiniteOnTheFlySolver` (see Section 10.3). Other quantities that remain constant for all values of k and n include: the number of initial states (4), the number of accepting states (2), and the maximum number of patches in the denotation of any state (1). The maximum recursive depth reported in Table 12.3 indicate the maximum recursive depth of the on-the-fly algorithm. Analysing the execution times reported in Table 12.2, we observe that the tool performs efficiently when the total number of variables is limited to $n = 3$ or $n = 4$. In these cases, even for formulas with 10 alternations, the runtime remains well below one second. However, the situation drastically changes as the number of variables increases. For $n = 5$, execution time grows significantly with the alternation depth k, exceeding 5 seconds for $k = 10$. The impact becomes even more pronounced when $n = 6$, where the time

Table 12.2: Execution time (in seconds) for each combination of alternation depth k and number of variables n.

n k	$n = 3$	$n = 4$	$n = 5$	$n = 6$
$k = 1$	0.050	0.090	0.590	264.962
$k = 2$	0.054	0.105	1.426	366.610
$k = 3$	0.057	0.123	1.909	446.490
$k = 4$	0.067	0.129	2.494	514.778
$k = 5$	0.075	0.150	2.990	634.879
$k = 6$	0.073	0.161	3.619	706.207
$k = 7$	0.072	0.172	4.118	922.110
$k = 8$	0.084	0.174	4.746	1015.996
$k = 9$	0.094	0.199	5.233	1073.501
$k = 10$	0.083	0.207	5.823	1158.499

Table 12.3: Values constant across variable count n, shown for increasing alternation depth k.

k	States	Edges	Denot Calls	Max Rec. Depth	Reach Calls
1	14	22	12	29	20
2	18	30	16	37	27
3	22	38	20	45	34
4	26	46	24	53	41
5	30	54	28	61	48
6	34	62	32	69	55
7	38	70	36	77	62
8	42	78	40	85	69
9	46	86	44	93	76
10	50	94	48	101	83

required for solving the denotation problem exceeds 1000 seconds in the last three cases. This exponential growth in computation time highlights the intrinsic complexity of the problem and its sensitivity to both the size of the formula (i.e., the number of alternations) and the dimensionality of the polyhedral system. A deeper analysis of the results, taking into account the statistics reported in Table 12.3, reveals that the underlying automaton remains relatively compact across all experiments. Specifically, the number of states and transitions never exceeds 100, even for formulae with up to 10 alternations. Consequently, the number of recursive calls and invocations of the `denot()` function also remains limited, as does the total number of calls to the *reach* operators. Despite this, the total execution time grows significantly, as seen in Table 12.2. This strongly suggests that the primary computational bottleneck lies in the evaluation of the *reach* operators. These observations highlight the need to optimise the implementation of reachability operators to improve scalability.

12.3 RTL Automaton Visualisation

Figure 12.17 shows the automaton constructed for the formula:

$$t0 \wedge \mathsf{G}(t1) \wedge \mathsf{F}(p \wedge \mathsf{F}(q)) \,.$$

The automaton consists of 15 states and 24 transitions. Double-circled states are accepting states. This automaton is an instance of a `BackwardNFA`, meaning it is the transposed graph of a standard nondeterministic finite automaton. The node labelled *Dummy Initial State* is a fake initial state with outgoing transitions to all accepting states. The on-the-fly algorithm `DenotOnTheFly` begins exploration from the accepting states and proceeds backward toward the initial ones. Each state is associated with a denotation, and in this specific case, each denotation consists of a single patch. These denotations are derived from propositional edge formulae (see Section 9.2.2). Transitions in Figure 12.17 are labelled with conjunctions of atomic propositions that appear in the formula used to compute the denotation of the target state. The atomic proposition *sing* alternates between one transition and another. Final states are labelled with *sing* (as seen from incoming transitions), while initial states are those with no incoming transitions.

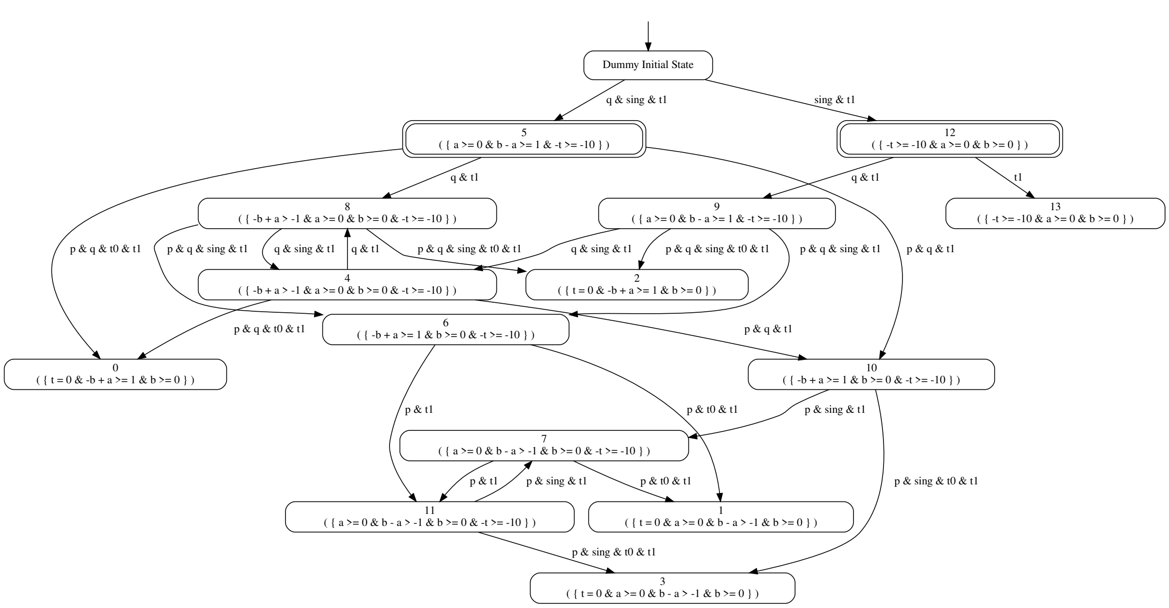

[Büchi]
Dummy Initial State
q & sing & t1
sing & t1
5
({ a >= 0 & b - a >= 1 & -t >= -10 })
12
({ -t >= -10 & a >= 0 & b >= 0 })
q & t1
q & t1
t1
8
({ -b + a > -1 & a >= 0 & b >= 0 & -t >= -10 })
9
({ a >= 0 & b - a >= 1 & -t >= -10 })
13
({ -t >= -10 & a >= 0 & b >= 0 })
p & q & t0 & t1
p & q & sing & t1
q & sing & t1
q & t1
p & q & sing & t0 & t1
q & sing & t1
p & q & sing & t0 & t1
p & q & sing & t1
p & q & t1
4
({ -b + a > -1 & a >= 0 & b >= 0 & -t >= -10 })
2
({ t = 0 & -b + a >= 1 & b >= 0 })
p & q & t0 & t1
6
({ -b + a >= 1 & b >= 0 & -t >= -10 })
p & q & t1
0
({ t = 0 & -b + a >= 1 & b >= 0 })
10
({ -b + a >= 1 & b >= 0 & -t >= -10 })
p & t1
p & t0 & t1
p & sing & t1
7
({ a >= 0 & b - a > -1 & b >= 0 & -t >= -10 })
p & t1
p & sing & t1
p & t0 & t1
p & sing & t0 & t1
11
({ a >= 0 & b - a > -1 & b >= 0 & -t >= -10 })
1
({ t = 0 & a >= 0 & b - a > -1 & b >= 0 })
p & sing & t0 & t1
3
({ t = 0 & a >= 0 & b - a > -1 & b >= 0 })

Figure 12.17: BackwardNFA for the formula $t0 \wedge \mathsf{G}(t1) \wedge \mathsf{F}(p \wedge \mathsf{F}(q))$.

12.4 Profiling On-the-Fly Execution: Fine-Grained Statistics

In this section, we leverage the built-in statistics engine described in Section 11.4 to collect detailed profiling data on the execution of the on-the-fly algorithm. The experiment is based on the polyhedral system introduced in Section 11.2.1, consisting of two tanks and a global clock variable t with constant derivative $\dot{t} = 1$, yielding a three-dimensional state space. The `RTL` formula used is $\varphi_k^{p,q}$, which expresses an alternating sequence of observations between atomic propositions p and q, with a fixed spatial gap of 1. The maximum allowed trajectory duration is 10 time units. For each alternation depth k, we compute the existential denotation under finite-time semantics. Table 12.4 reports all collected metrics for increasing values of k, starting from $k = 1$ up to $k = 260$, in increments of 10. The meaning of each column is self-explanatory:

- **Paths**: total number of execution paths explored by the algorithm to collect new contributions to the final result;
- **Redundant Paths**: number of paths that yielded contributions already encountered in previous explorations.

A hard limit of 10.000 calls to the `denot()` function was enforced. Whenever this threshold is exceeded, the solver terminates early, possibly yielding an incomplete result. Rows highlighted in red indicate executions where this limit was reached. Notably, these scenarios correspond to configurations where the maximum number of patches per state increases to 2. This directly impacts the algorithm's complexity: the number of possible paths grows exponentially with the number of patches per state. Even though not explicitly shown in Table 12.4, starting from $k = 254$, about half of the automaton states have two patches. One of the most significant indicators of inefficiency is the rapid increase in the number of redundant paths. For instance, when $k = 254$, the number of total paths explored is 3701, but only 12 of these contribute new information, meaning that 3689 paths are redundant. This observation highlights a clear bottleneck in the current implementation: an excessive number of redundant computations. Therefore, strategies aimed at reducing redundant path exploration, such as memoization or early stopping heuristics may drastically improve performance.

Table 12.4: Performance metrics at increasing alternation depth k for $n = 3$ variables.

k	Denot Time (s)	Denot Calls	Reach Calls	Paths	Redund. Paths	States	Edges	Patches	Max Patches	Max Rec. Depth	SCCs
1	0.004500	12	20	2	0	14	22	14	1	29	13
10	0.028652	48	83	2	0	50	94	50	1	101	40
20	0.057424	88	153	2	0	90	174	90	1	181	70
30	0.080334	128	223	2	0	130	254	130	1	261	100
40	0.106735	168	293	2	0	170	334	170	1	341	130

Table continues on next page

Table 12.4 continued from previous page

k	Denot Time (s)	Denot Calls	Reach Calls	Paths	Redund. Paths	States	Edges	Patches	Max Patches	Max Rec. Depth	SCCs
50	0.132597	208	363	2	0	210	414	210	1	421	160
60	0.276808	495	867	4	0	252	500	252	1	505	191
70	0.333177	575	1007	4	0	292	580	292	1	585	221
80	0.374905	655	1147	4	0	332	660	332	1	665	251
90	0.424215	735	1287	4	0	372	740	372	1	745	281
100	0.466706	815	1427	4	0	412	820	412	1	825	311
110	0.518713	895	1567	4	0	452	900	452	1	905	341
120	0.570648	975	1707	4	0	492	980	492	1	985	371
130	0.596277	1055	1847	4	0	532	1060	532	1	1065	401
140	0.673178	1135	1987	4	0	572	1140	572	1	1145	431
150	0.723608	1215	2127	4	0	612	1220	612	1	1225	461
160	0.771126	1295	2267	4	0	652	1300	652	1	1305	491
170	0.807296	1375	2407	4	0	692	1380	692	1	1385	521
180	0.863785	1455	2547	4	0	732	1460	732	1	1465	551
190	0.887465	1535	2687	4	0	772	1540	772	1	1545	581
200	0.949372	1615	2827	4	0	812	1620	812	1	1625	611
210	1.017950	1695	2967	4	0	852	1700	852	1	1705	641
220	1.063648	1775	3107	4	0	892	1780	892	1	1785	671
230	1.119252	1855	3247	4	0	932	1860	932	1	1865	701
240	1.157979	1935	3387	4	0	972	1940	972	1	1945	731
250	1.209950	2015	3527	4	0	1012	2020	1012	1	2025	761
251	1.220496	2023	3541	4	0	1016	2028	1016	1	2033	764
252	1.226105	2031	3555	4	0	1020	2036	1020	1	2041	767
253	1.227202	2039	3569	4	0	1024	2044	1024	1	2049	770
254	18.164566	10001	10361	3701	3689	1028	2052	1538	2	3077	773
255	14.224034	10001	10363	3700	3687	1032	2060	1544	2	3089	776
256	17.186450	10001	10363	3700	3685	1036	2068	1550	2	3101	779
257	15.150799	10001	10362	3700	3686	1040	2076	1556	2	3113	782
258	18.505702	10001	10368	3698	3686	1044	2084	1562	2	3125	785
259	15.280793	10001	10367	3698	3687	1048	2092	1568	2	3137	788
260	18.873083	10001	10372	3697	3685	1052	2100	1574	2	3149	791

Part IV

Conclusion

–13–

Conclusions and Future Work

This thesis presented the design, formalisation, and implementation of `rtl-mc` [26], a model-checking tool for verifying temporal properties of single-location linear hybrid systems. These systems are described through a formal specification language based on polyhedral systems, and verified against temporal logic formulae expressed in RTL (Real-Time Logic), a real-time extension of LTL suitable for continuous-time semantics. The main contribution of this work is a modular, extensible, and fully automated verification framework. The tool supports the entire verification pipeline: parsing polyhedral system specifications, translating RTL into LTL, generating automata representations, and performing model checking under multiple semantics—finite-time, infinite-time, may, and must. A detailed architectural breakdown has been provided, including the symbolic denotation algorithms that compute satisfaction sets.

The framework has been evaluated across a variety of experimental settings. We investigated how changes in temporal and spatial parameters affect the verification result. Further experiments assessed the tool's scalability with increasing system dimensionality and formula complexity. Fine-grained profiling revealed bottlenecks, especially in the reachability computations, and highlighted performance limitations of the current implementation. Overall, results show that the tool can effectively handle moderately sized single-location systems, providing sound results with acceptable computational overhead. Additionally, the built-in statistics engine offers valuable diagnostic information that can guide optimisation and solver selection.

Future Work

While this thesis lays a solid foundation for the symbolic verification of polyhedral systems, several directions for future improvement and research are identified:

- **Scalability Enhancements:** Experimental results highlight that performance degrades rapidly with increasing system dimensionality. The main computational bottleneck lies in the reachability operators, which rely on the Parma Polyhedra Library (PPL). Investigating alternative geometric libraries or developing custom optimisations for PPL could significantly improve performance.

- **Parallel and Concurrent Execution:** Although a preliminary concurrent execution mode has been introduced (see Section A.5.4), it remains unstable and limited by the lack of thread safety in PPL. A redesign of the solver to support safe and efficient parallelism would allow better exploitation of modern multi-core architectures.
- **Reduction of Redundant Exploration:** As shown in Section 12.4, the on-the-fly solver often explores redundant paths that do not contribute to the final result. Incorporating caching strategies, memoisation, or early-pruning heuristics, while preserving completeness, could lead to significant performance gains.
- **Support for Multi-location Systems:** Extending the framework to support systems with multiple modes and discrete transitions would broaden its applicability to full-fledged linear hybrid automata.
- **Graphical Interface and Visualisation Tools:** Providing a graphical user interface would improve usability, especially for non-expert users. Visual representations of automata, polyhedral regions, trajectories, and verification results would facilitate interpretation and debugging.

Final Remarks

By integrating formal definitions, symbolic abstractions, and practical implementation, this work delivers a verification platform that bridges theoretical research and applied tool development in hybrid systems verification. The architecture, algorithms, and semantics introduced are general enough to be adapted and extended to related domains in formal methods and cyber-physical systems.

As the demand for the formal verification of cyber-physical systems grows, tools like `rtl-mc` [26] have the potential to play a central role in bridging the gap between theoretical research and practical system validation.

–A–

Command-Line Interface Reference for `rtl-mc`

This appendix documents all command-line options available for the `rtl-mc` tool, which performs model checking of `RTL` formulae on polyhedral systems. Options are grouped by category for clarity.

A.1 Input Specification Options

A.1.1 Polyhedral System Input

- `-sf`, `--system-file` <*FILE*>
  Load the polyhedral system from the specified file.

- `-ss`, `--system-string` <*STRING*>
  Provide the polyhedral system directly as a string.

- `-si`, `--system-stdin`
  Read the polyhedral system from standard input (default).

A.1.2 `RTL` Formula Input

- `-ff`, `--formula-file` <*FILE*>
  Load the `RTL` formula from the specified file.

- `-fs`, `--formula-string` <*STRING*>
  Provide the `RTL` formula directly as a string.

- `-fi`, `--formula-stdin`
  Read the formula from standard input (default).

A.2 Verification Semantics Options

A.2.1 Trajectory Semantics

- `--semantics=`*`VAR`*
  Specify the type of trajectories considered during verification:

  `fin` Finite-time trajectories (default). Useful for reachability and bounded-time properties.

  `inf` Infinite-time trajectories. Suitable for liveness and fairness properties.

  `may` Trajectories that are either infinite or terminate at a *may-exit* point, i.e. a boundary point from which at least one admissible direction exits the invariant.

  `must` Trajectories that are either infinite or terminate at a *must-exit* point, i.e. a boundary point from which all admissible directions exit the invariant.

A.2.2 Verification Modes: Existential or Universal

- `--existential`
  Compute the existential denotation: all points from which *at least one* trajectory satisfies the formula φ (default).

- `--universal`
  Compute the universal denotation: all points from which *every* trajectory satisfies φ.

A.2.3 Model-Checking Problem

- `--mc` <*POINT*>
  Check whether a specific point $x \in \mathbb{Q}^n$ satisfies the formula φ. The format is a list of rational assignments for each system variable, e.g., `[x=1/2, y=-1/3]`. This option is evaluated with respect to the chosen semantics and verification mode (existential or universal).

A.3 Automaton Formula Translation Optimisations

These options control the Spot translation of discretised `LTL` formulae into automata.

- `--low`
  Use minimal Spot optimisations for faster but potentially larger automata (default).

- `--medium`
  Apply moderate Spot optimisations.

- `--high`
  Enable all Spot optimisations (slower, but may reduce automaton size).
- `--any`
  No preference. Let Spot choose the strategy.

A.4 Output Format

A.4.1 Verbosity Levels

- `-V`, `--verbose`
  Enable verbose output. This option can be repeated (e.g., `-VVV`) to increase verbosity level.
- `-q`, `--quiet`
  Suppress all normal output.

A.4.2 Statistics Engine

- `-s`, `--stats` <*FORMAT*>
  Output a formatted summary of performance statistics. The format depends on the internal statistics engine configuration. See Section 11.4.

A.5 Advanced Options

A.5.1 Exporting all Automata in DOT Format

- `--export-automaton-dot`
  Export all generated automata (including intermediate automata) to DOT files for visualisation with Graphviz.

A.5.2 Exporting On-The-Fly Algorithm Execution in DOT Format

- `--export-denot-path-tree-dot`
  Export the tree of execution paths explored by the on-the-fly algorithm as DOT files.

A.5.3 Direct Discretisation of `RTL` Formulae

- `--direct-ltl`
  Discretise the input `RTL` formula directly into LTL in a single step, bypassing intermediate structures. This is an experimental option and is only valid under finite-time semantics.

A.5.4 Concurrent Execution Mode

- `-c`, `--concurrent`
  Enable concurrent execution of the on-the-fly algorithm. This feature is highly experimental and is only supported under finite-time semantics.

A.6 Special Environment Variables

The execution of `rtl-mc` can be influenced by setting specific environment variables. Currently, only one environment variable is supported:

`DENOT_MAX` Sets an upper bound on the number of invocations of the `denot()` function during the on-the-fly algorithm. Once this threshold is reached, the algorithm terminates early and returns an *incomplete* result. This feature prevents excessive computation in cases of combinatorial explosion.
Default value: `100000`.
If the execution exceeds this threshold, the following warning message is printed:

```
WARNING: The result is incomplete because the solver reached
the maximum number of iterations (can be configured via
DENOT_MAX environment variable, default value: 100000)
```

Bibliography

[1] Thomas A. Henzinger. “The Theory of Hybrid Automata”. In: (2000). Ed. by M. Kemal Inan and Robert P. Kurshan, pp. 265–292. DOI: `10.1007/978-3-642-59615-5_13`. URL: `https://doi.org/10.1007/978-3-642-59615-5_13`.

[2] Hidde de Jong. “Modeling and Simulation of Genetic Regulatory Systems: A Literature Review”. In: *Journal of Computational Biology* 9.1 (2002). PMID: 11911796, pp. 67–103. DOI: `10.1089/10665270252833208`. eprint: `https://doi.org/10.1089/10665270252833208`. URL: `https://doi.org/10.1089/10665270252833208`.

[3] João P. Hespanha. “A model for stochastic hybrid systems with application to communication networks”. In: *Nonlinear Analysis: Theory, Methods Applications* 62.8 (2005). Hybrid Systems and Applications, pp. 1353–1383. ISSN: 0362-546X. DOI: `https://doi.org/10.1016/j.na.2005.01.112`. URL: `https://www.sciencedirect.com/science/article/pii/S0362546X05004074`.

[4] Jerry Ding et al. “Hybrid Systems in Robotics”. In: *IEEE Robotics Automation Magazine* 18.3 (2011), pp. 33–43. DOI: `10.1109/MRA.2011.942113`.

[5] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. “What’s Decidable about Hybrid Automata?” In: *Journal of Computer and System Sciences* 57.1 (1998), pp. 94–124. ISSN: 0022-0000. DOI: `https://doi.org/10.1006/jcss.1998.1581`. URL: `https://www.sciencedirect.com/science/article/pii/S0022000098915811`.

[6] Gerd Behrmann, Alexandre David, and Kim G. Larsen. “A Tutorial on Uppaal”. In: *Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Bertinora, Italy, September 13-18, 2004, Revised Lectures*. Ed. by Marco Bernardo and Flavio Corradini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 200–236. ISBN: 978-3-540-30080-9. DOI: `10.1007/978-3-540-30080-9_7`. URL: `https://doi.org/10.1007/978-3-540-30080-9_7`.

[7] Thomas A. Henzinger, P-H. Ho, and H. Wong-Toi. *HYTECH: A Model Checker for Hybrid Systems*. Tech. rep. UCB/ERL M97/79. Oct. 1997. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/1997/3324.html`.

[8] Goran Frehse et al. “SpaceEx: Scalable Verification of Hybrid Systems”. In: (2011). Ed. by Ganesh Gopalakrishnan and Shaz Qadeer, pp. 379–395.

[9] Massimo Benerecetti and Marco Faella. “Automatic Synthesis of Switching Controllers for Linear Hybrid Systems: Reachability Control”. In: *ACM Trans. Embed. Comput. Syst.* 16.4 (May 2017). ISSN: 1539-9087. DOI: `10.1145/3047500`. URL: `https://doi.org/10.1145/3047500`.

[10] Goran Frehse. “PHAVer: algorithmic verification of hybrid systems past HyTech”. In: *Int. J. Softw. Tools Technol. Transf.* 10.3 (May 2008), pp. 263–279. ISSN: 1433-2779.

[11] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. “Flow*: An Analyzer for Non-linear Hybrid Systems”. In: *Computer Aided Verification*. Ed. by Natasha Sharygina and Helmut Veith. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 258–263. ISBN: 978-3-642-39799-8.

[12] The Ariadne Team. *Ariadne: an open library for formal verification of cyber-physical systems*. 2020. URL: `http://www.ariadne-cps.org` (visited on 04/21/2025).

[13] Stefan Schupp, Erika Ábrahám, Ibtissem Makhlouf, and Stefan Kowalewski. “HyPro: A C++ Library of State Set Representations for Hybrid Systems Reachability Analysis”. In: Apr. 2017, pp. 288–294. ISBN: 978-3-319-57287-1. DOI: `10.1007/978-3-319-57288-8_20`.

[14] Tommaso Dreossi. “Sapo: Reachability Computation and Parameter Synthesis of Polynomial Dynamical Systems”. In: *Proceedings of the 20th International Conference on Hybrid Systems: Computation and Control*. HSCC ’17. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 2017, pp. 29–34. ISBN: 9781450345903. DOI: `10.1145/3049797.3049824`. URL: `https://doi.org/10.1145/3049797.3049824`.

[15] Massimo Benerecetti, Marco Faella, and Fabio Mogavero. “Model Checking Linear Temporal Properties on Polyhedral Systems”. In: *Leibniz International Proceedings in Informatics (LIPIcs)* 318 (2024), 19:1–19:23. ISSN: 1868-8969. DOI: `10.4230/LIPIcs.TIME.2024.19`. URL: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TIME.2024.19`.

[16] Radha Poovendran. “Cyber–Physical Systems: Close Encounters Between Two Parallel Worlds [Point of View]”. In: *Proceedings of the IEEE* 98.8 (2010), pp. 1363–1366. DOI: `10.1109/JPROC.2010.2050377`.

[17] M. Reynolds. “The complexity of temporal logic over the reals”. In: *Annals of Pure and Applied Logic* 161.8 (2010), pp. 1063–1096. ISSN: 0168-0072. DOI: `https://doi.org/10.1016/j.apal.2010.01.002`. URL: `https://www.sciencedirect.com/science/article/pii/S0168007210000035`.

[18] Moshe Y. Vardi and Pierre Wolper. "Automata-theoretic techniques for modal logics of programs". In: *Journal of Computer and System Sciences* 32.2 (1986), pp. 183–221. ISSN: 0022-0000. DOI: `https://doi.org/10.1016/0022-0000(86)90026-7`. URL: `https://www.sciencedirect.com/science/article/pii/0022000086900267`.

[19] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear temporal logic and linear dynamic logic on finite traces". In: IJCAI '13 (2013), pp. 854–860.

[20] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. "The Parma Polyhedra Library: Toward a Complete Set of Numerical Abstractions for the Analysis and Verification of Hardware and Software Systems". In: *CoRR* abs/cs/0612085 (2006). arXiv: `cs/0612085`. URL: `http://arxiv.org/abs/cs/0612085`.

[21] Alexandre Duret-Lutz et al. *From Spot 2.0 to Spot 2.10: What's New?* 2022. arXiv: `2206.11366 [cs.LO]`. URL: `https://arxiv.org/abs/2206.11366`.

[22] Oded Maler and Dejan Nickovic. "Monitoring Temporal Properties of Continuous Signals". In: *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*. Ed. by Yassine Lakhnech and Sergio Yovine. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 152–166. ISBN: 978-3-540-30206-3.

[23] David Bremner. "Incremental Convex Hull Algorithms Are Not Output Sensitive". In: *Discrete & Computational Geometry* 21.1 (1999), pp. 57–68. ISSN: 1432-0444. DOI: `10.1007/PL00009410`. URL: `https://doi.org/10.1007/PL00009410`.

[24] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. 1977, pp. 46–57. DOI: `10.1109/SFCS.1977.32`.

[25] Stéphane Demri and Philippe Schnoebelen. "The Complexity of Propositional Linear Temporal Logics in Simple Cases". In: *Information and Computation* 174.1 (2002), pp. 84–103. ISSN: 0890-5401. DOI: `https://doi.org/10.1006/inco.2001.3094`. URL: `https://www.sciencedirect.com/science/article/pii/S0890540101930949`.

[26] Vincenzo Tramo. *rtl-mc: A Model Checker for Real-Time Logic on Polyhedral Systems*. `https://github.com/vtramo/rtl-mc`. Accessed: 2025-05-19. 2025.

[27] *Meson Build System*. Accessed: 2025-05-07. URL: `https://mesonbuild.com`.

[28] Freedesktop.org. *pkg-config*. `https://www.freedesktop.org/wiki/Software/pkg-config/`. Accessed: 2025-05-07. 2024.

[29] Massimo Benerecetti, Marco Faella, and Stefano Minopoli. "Towards Efficient Exact Synthesis for Linear Hybrid Systems". In: *Electronic Proceedings in Theoretical Computer Science* 54 (June 2011), pp. 263–277. ISSN: 2075-2180. DOI: `10.4204/eptcs.54.19`. URL: `http://dx.doi.org/10.4204/EPTCS.54.19`.

[30] Jørn Lind-Nielsen. *BuDDy: A Binary Decision Diagram Package*. Accesso: maggio 2025. Technical University of Denmark. 1999. URL: `https://buddy.sourceforge.net/manual/`.

[31] Spot Team. *Spot's Temporal Logic Formulas*. Accessed: 2025-05-11. 2025. URL: `https://spot.lre.epita.fr/tl.pdf`.

[32] Sonali Dutta and Moshe Y. Vardi. "Assertion-based flow monitoring of SystemC models". In: *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*. 2014, pp. 145–154. DOI: `10.1109/MEMCOD.2014.6961853`.

[33] Terence Parr. *The Definitive ANTLR 4 Reference*. 2nd. Pragmatic Bookshelf, 2013. ISBN: 1934356999.

[34] GeoGebra Team. *About GeoGebra*. `https://www.geogebra.org/about`. Accessed: 2025-05-18. GeoGebra, 2025.

List of Figures

2.1 A convex polyhedron $P \subset \mathbb{R}^2$ defined as the intersection of five half-spaces. The dashed edges correspond to strict inequalities and are therefore not part of the polyhedron. The solid edges, corresponding to non-strict inequalities, are included in P. 10
2.2 Representation of a General Polyhedra through Simple Convex Polyhedra 11
2.3 Effect of the closure operation: boundary points excluded in the original polyhedron P (a) are included in its closure $cl(P)$ (b). 12
2.4 Comparison between a polyhedron defined with non-strict constraints (a) and its interior (b). 13
2.5 Illustration of point-to-polyhedron adjacency. Points p_1, p_2, and p_3 are adjacent to the convex polyhedron P, either because they lie within P or on its (possibly open) boundary. Point p_4 is not adjacent, as it lies outside the closure of P. 14
2.6 Examples of adjacency and non-adjacency between two convex polyhedra P and Q, based on the definition of their border. 15
2.7 Comparison between a closed unbounded polyhedron (a) and a compact polyhedron (b). 16
2.8 Visualization of atomic propositions (a) and the induced partition (b). 18
2.9 A flow constraint *Flow* and a cone of directions from a point p 19
2.10 Geometric intuition of the positive pre-flow $P_{\swarrow_{>0}}$. 25

5.1 Components of the system and their interactions, illustrating the high-level architecture. 54
5.2 Visual legend for class diagrams. 55
5.3 `cli` Analysis Class Diagram 56
5.4 `parser` Analysis Class Diagram 56
5.5 `system` Analysis Class Diagram 57
5.6 `abstractions` Analysis Class Diagram 58
5.7 `discretisation` Analysis Class Diagram 58
5.8 `automata` Analysis Class Diagram 59
5.9 `mcsolvers` Analysis Class Diagram 61

6.1 Example of a convex polyhedron represented both as a system of linear constraints (top-right) and as a generator system (bottom-right), consisting of two points and two rays. 71
6.2 Architecture of the Spot Project 76
6.3 Tree representation of the LTL formula $\mathsf{F}(p \wedge \mathsf{F}(q))$ constructed using `spot::formula`. 77
6.4 Transition-Based ω-Automaton 80
6.5 State-Based ω-Automaton 80
6.6 Automaton resulting from the translation of the LTL formula $F(p \wedge F(q))$. 81

7.1 `PolyhedralSystemParsingResult` class 89
7.2 `PolyhedralSystem` class 90
7.3 `AtomInterpretation` class 91
7.4 `Observable` class 96
7.5 `PolyhedralSystemSymbolTable` class 98
7.6 `system` component class diagram 102

8.1 `RtlParsingResult` class 106
8.2 `DiscreteFiniteLtlFormula` and `DiscreteLtlFormula` classes 113

9.1 `Automaton` class 116
9.2 `AutomatonAcceptance` class 116
9.3 `PolyhedralLtlAutomaton` class 119
9.4 `StateDenotation` class 121
9.5 `PolyhedralSystemFormulaDenotationMap` class 121
9.6 Concrete Polyhedral Automata Classes 123
9.7 `PolyhedralAbstraction` class 124
9.8 `omnidirectional` component class diagram 125
9.9 `PolyhedralSynchronousProductAutomaton` class 126
9.10 `DenotOnTheFly` class 128

10.1 `Solver` abstract class 132
10.2 `ExplicitSolver` abstract class 133
10.3 Hierarchy of omnidirectional solvers 134
10.4 Class hierarchy of on-the-fly solvers. 136
10.5 Solver selection diagram for `fin` and `inf` semantics. 137
10.6 Solver selection diagram for `may` and `must` semantics. 137

11.1 Generalised GAP experiment with n tanks 146

11.2 Visualisation of the existential denotation for the GAP experiment with 2 tanks and a time bound of 10. The red region corresponds to the set of initial configurations from which the system satisfies the finite-time formula $t_0 \wedge \mathsf{G}(t_1) \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q \wedge \mathsf{F}(p \wedge \mathsf{F}(q)))))$. The computed result is the polyhedron: `( { t = 0 & b ≥ 0 & a ≥ 0 & -b + a ≥ -23 & b - a ≥ -21 } )`. The clock variable t is fixed to 0 and eliminated by projection in the final visualisation. 152

12.1 Visual representations of the flow constraints used in the GAP experiments. 158
12.2 Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $Flow^{\text{std}}$. 162
12.3 Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $Flow^{\text{std}}$. 162
12.4 Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under may semantics and $Flow^{\text{std}}$. 163
12.5 Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under may semantics and $Flow^{\text{std}}$. 163
12.6 Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $Flow^{\text{rect}}$. 164
12.7 Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under finite-time semantics and $Flow^{\text{rect}}$. 164
12.8 Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ under may semantics and $Flow^{\text{rect}}$.165
12.9 Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 15$ under may semantics and $Flow^{\text{rect}}$.165
12.10Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 10$ under finite-time semantics and $Flow^{\text{up}}$. 165
12.11Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ under finite-time semantics and $Flow^{\text{up}}$. 166
12.12$\varphi^{p,q}_{\text{gap}=1}$ and $\varphi^{q,p}_{\text{gap}=1}$ of δ under all semantics and $Flow^{\text{std}}$ without clock variable (omnidirectional flow). 166
12.13Formula $\varphi^{p,q}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ under may semantics and $Flow^{\text{must}}$. 166
12.14Formula $\varphi^{q,p}_{\text{gap}=\delta}$ from $\delta = 1$ to $\delta = 5$ under may semantics and $Flow^{\text{must}}$. 166
12.15Formula $\varphi^{p,q}_{\text{gap}=1}$ with max time from $t = 1$ to $t = 10$ under finite-time semantics and $Flow^{\text{std}}$. 167
12.16Admissible trajectories from selected initial points under may-semantics, formula $\varphi^{p,q}_{\text{gap}=1}$ and $Flow^{\text{rect}}$ 169
12.17BackwardNFA for the formula $t0 \wedge \mathsf{G}(t1) \wedge \mathsf{F}(p \wedge \mathsf{F}(q))$. 173

List of Tables

2.1 Notation used in the thesis: three types of trace-like objects (from the most concrete to the most abstract) and the time decomposition of a signal. 23

5.1 Functional Requirements of the `RTL` Model Checking Tool 52
5.2 Non-Functional Requirements of the `RTL` Model Checking Tool 53

6.1 Common Meson commands 65
6.2 Type aliases defined in `ppl_aliases.h` for core PPL types. 72
6.3 Main operations on single convex polyhedra (`Poly`) and their PPL methods. 73
6.4 Utility functions in `PPLUtils` with their signatures and descriptions. . 75
6.5 Logical Operators Syntax 79
6.6 Temporal Operators Syntax 79

7.1 Functions for parsing a polyhedral system specification. 87
7.2 Methods for accessing atomic propositions and their interpretations in the `PolyhedralSystem` class. 92
7.3 Methods for accessing and analyzing the invariant region in the `PolyhedralSystem` class. 93
7.4 Methods for accessing and analysing the flow properties in the `PolyhedralSystem` class. 94
7.5 Methods for generating observables in the `PolyhedralSystem`. 96
7.6 Methods for interacting with the `PolyhedralSystemSymbolTable`. . . 100

8.1 Functions for parsing RTL formulae in the `parsertl.h` module. . . . 105
8.2 Basic logical constants and atomic propositions. 107
8.3 Functions for creating temporal logic formulas. 108
8.4 Functions for conjunction, disjunction, negation, and atomic propositions. 109
8.5 Functions for handling the *sing* atomic proposition. 109
8.6 Functions for collecting and checking properties of atomic propositions and formulas. 110
8.7 Main methods of `DiscreteLtlFormula` and `DiscreteFiniteLtlFormula` classes 113

9.1 Public methods of the `Automaton` class. 117
9.2 Methods of the `PolyhedralLtlAutomaton` class. 118

11.1 Statistics for RTL formulae, polyhedral systems, and discretisation. . 153
11.2 Statistics related to automaton construction, translation, optimisation, and patches. 154
11.3 Statistics for NFA and on-the-fly algorithm. 155

12.1 Atomic propositions and formulae used in the GAP model checking experiments. 158
12.2 Execution time (in seconds) for each combination of alternation depth k and number of variables n. 171
12.3 Values constant across variable count n, shown for increasing alternation depth k. 171
12.4 Performance metrics at increasing alternation depth k for $n = 3$ variables. 174