Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

The present work was submitted to the LuFG Theory of Hybrid Systems

BACHELOR OF SCIENCE THESIS

EXTENDING THE REALYST TOOL TO SUPPORT LINEAR HYBRID AUTOMATA

Dilara Arslan

Communicated by
Prof. Dr. Erika Ábrahám

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Thomas Noll

Additional Advisor:
József Kovács

Aachen, 05.05.2026

Abstract

Hybrid automata are widely used to model systems that combine discrete and continuous behaviour. In addition, stochastic hybrid automata extend these models by including probabilistic transitions, enabling the analysis of systems with uncertainty. The analysis of these automata can be done using different tools such as RealySt or Modest. However, these tools differ in their internal representation, which makes the exchange of models incompatible. Therefore, the Jani specification was introduced to provide a common format for exchanging models. In this thesis, we extend the parser between the Jani specification and the tool RealySt. Previous versions of the parser were restricted to rectangular hybrid automata and did not support linear hybrid automata. Since linear hybrid automata are more expressive, we enhanced the parser to support linear expressions in flows, guards, and invariants, as well as resets. Furthermore, we extended the handling of specifications during the export from RealySt to Jani. We evaluated our extension by testing the parser with test files and plotting the outputs with Modest to check the compatibility with another tool. We found out that our extensions to parse linear hybrid automata are compatible with Modest, but our extension to parse goal locations in a specification is not.

Contents

Chapter 1

Introduction

Stochastic hybrid automata combine discrete, continuous, and stochastic behaviour within a single modelling framework. Such models can be used to represent both physical phenomena and digital control systems. This model-based representation enables the use of formal methods in order to analyse system behaviour and compute probabilities of events [DSÁR23]. However, interactions between the continuous evolution of variables, discrete transitions between locations, and stochastic effects make the analysis of stochastic hybrid automata challenging [BDG+11].
Continuous behaviour can be described using intervals, constant rates, or linear dynamics. An example of continuous behaviour described by linear dynamics is given by heating systems or thermostat models, in which the temperature evolves continuously and depends linearly on the current state [Ábr21].
For modelling such systems, automata with linear time-invariant behaviour are suitable, as they provide a realistic representation of continuous dynamics while still remaining tractable for analysis [LG09]. These automata can be analysed using a variety of methods. In practice, however, these tools often rely on incompatible internal representations of hybrid automata, which complicates the model interchange and outcome comparison. To address this issue, the `jani-model` specification provides a common format to define hybrid automata [BDH+17], while tools such as RealySt [AG 25] offer a framework for modeling and analysing stochastic hybrid automata. Previous versions of RealySt were limited to rectangular flows, restricting the class of systems that could be modeled [DSSR23]. Compared to this, automata with linear time-invariant behaviour are more expressive than rectangular automata [Ábr21].
This thesis helps to bridge this gap by extending an existing parser between Jani and RealySt. In particular, the parser is enhanced to support linear time-invariant expressions, including linear flows, guards, invariants, and affine resets. This enables the representation of more realistic system dynamics, such as those arising in heating processes. Furthermore, the handling of specifications is improved by incorporating goal locations and time-bounded properties. The remainder of this thesis is structured as follows. Chapter 2 introduces the formal foundations relevant to stochastic hybrid systems and the Jani specification. Chapter 3 describes the technical background, including the tools and libraries used. Chapter 4 provides further details on the Jani specification. Chapter 5 presents the implementation details of the parser and its extensions. Chapter 6 summarizes the results and discusses the achieved functionality. Finally, Chapter 7 concludes the thesis and outlines directions for future work.

Chapter 2

Preliminaries

This chapter introduces the basic concepts used throughout this thesis, including hybrid automata, linear hybrid automata, linear hybrid automata with random clocks, and reachability analysis.

2.1 Hybrid Automata

A hybrid system combines the characteristics of discrete and continuous systems. The transitions in hybrid systems are discrete, but the evolution of variables is continuous and described through ordinary differential equations. Hybrid systems can be used to model processes in the real world [Sch19].
Hybrid automata are formal models for hybrid systems that combine discrete and continuous dynamics. They extend automata by introducing variables that evolve continuously over time according to specified flows. In addition to discrete transitions between locations, which may be restricted by guards and resets, hybrid automata capture the interaction between discrete changes and continuous evolution [Ábr21]. In this paper, the definitions and notations of [WRÁ24] are followed, as they align with the internal representation used in RealySt and match the structure required by the tool.

Definition 2.1.1 (Hybrid Automata Syntax). A hybrid automaton is a tuple $H = (Loc, Var, Flow, Inv, Lab, Edge, Init)$ where

- Loc is a non-empty and finite *set of locations*.
- $Var = \{x_1, ..., x_d\}$ is a finite ordered *set of variables*.
- $Flow : Loc \to (\mathbb{R}^d \to \mathbb{R}^d)$ defines the *flows or dynamics* of a location.
- $Inv : Loc \to 2^{\mathbb{R}^d}$ specifies an *invariant* for each location.
- $Lab = \{a_1,...,a_k\}$ is a non-empty finite ordered *set of labels*.
- $Edge \subseteq Loc \times Lab \times 2^{\mathbb{R}^d} \times (\mathbb{R}^d \to \mathbb{R}^d) \times Loc$ is a finite set of discrete *transitions or jumps*. For a jump $(\ell, a, g, r, \ell') \in Edge$, ℓ and ℓ' are its source respectively target locations, a is its label, g its *guard*, and r its *reset*.

- $Init : Loc \to 2^{\mathbb{R}^d}$ defines *initial valuations* for each location. We call a state $(\ell, \nu) \in \Sigma$, where Σ is the set of states, initial if $\nu \in Inv(\ell) \cap Init(\ell)$.

The state of a d-dimensional hybrid automaton $\sigma = (l, v) \in Loc \times \mathbb{R}^d$ contains a location and the variable distribution. During a time step, the values of the variables evolve depending on the flow of the current location. The flows define the derivatives of the variables. For a variable x with its derivative $\dot{x} = c$, where c is an arbitrary constant, the new value of x is $x + c$ after one time step. The time in a location progresses as long as the invariant of the location is satisfied or a discrete step (jump) is taken. A jump $(l, a, g, r, l') \in Edge$ describes the transition from location l to location l'. Two jumps with the same label cannot be executed simultaneously [WRÁ24].

Example 2.1.2 (Hybrid Automata Heat).

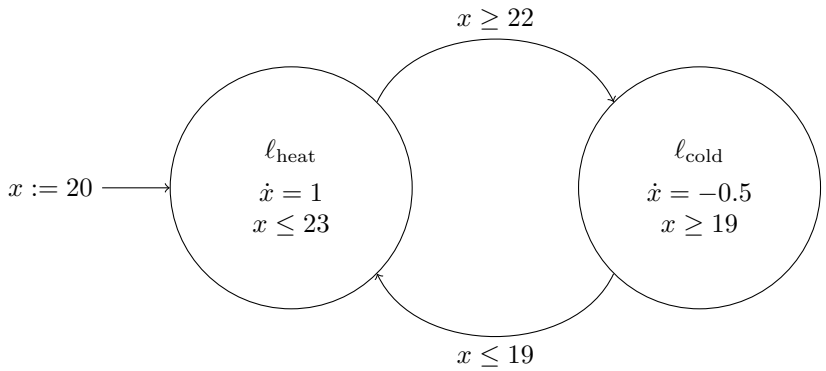

Figure 2.1: Example Heating Systems as an Automata.

In this example, inspired by [Ábr21], a heating system is modeled by a hybrid automaton. The example illustrates how continuous dynamics and discrete transitions interact in a simple real-world scenario. In particular, it demonstrates how flows describe the evolution of a variable over time, while guards restrict when transitions between locations can occur. The variable x specifies the temperature in a unit of temperature, for example ℃. The heating has two locations, as it can be on (l_{heat}) or off (l_{cold}). While the heating is working, the temperature increased by 1℃ in one time step, and it can only stay activated until the temperature reaches 23℃. If it is higher, the heater is turned off, which is symbolized by the transition from the location l_{heat} to l_{cold}. The transition has a guard, which defines that the transition is only allowed to take place, if the temperature is greater than or equal to 22℃.

Definition 2.1.3 (Hybrid Automata Semantics). The operational semantics of a hybrid automaton define its state transitions. A state can evolve either continuously over time according to the flow within a location or discretely by taking a transition to another location [DSÁR23]. For a d-dimensional hybrid automaton $H = (Loc, Var, Flow, Inv, Lab, Edge, Init)$, the operational semantics can be de-

fined by the following rules from [WRÁ24]:

$$\frac{\ell \in Loc \qquad \nu,\nu' \in \mathbb{R}^d \qquad t \in \mathbb{R}_{\geq 0} \qquad f : [0,t] \to \mathbb{R}^d \qquad \frac{df}{dt} = \dot{f} : (0,t) \to \mathbb{R}^d \qquad f(0) = \nu \qquad f(t) = \nu' \qquad \forall t' \in (0,t).\ \dot{f}(t') = Flow(\ell)(f(t')) \qquad \forall t' \in [0,t].\ f(t') \in Inv(\ell)}{(\ell,\nu) \xrightarrow{t} (\ell,\nu')} \text{ FLOW}$$

$$\frac{(\ell,a,g,r,\ell') \in Edge \qquad \nu,\nu' \in \mathbb{R}^d \qquad \nu \in g \qquad \nu' = r(\nu) \qquad \nu' \in Inv(\ell')}{(\ell,\nu) \xrightarrow{a} (\ell',\nu')} \text{ JUMP}$$

For a hybrid automaton H, a path π defines a finite or infinite sequence of alternating time or discrete steps. $\pi = \sigma_0 \xrightarrow{t_0} \sigma_0' \xrightarrow{a_{w(0)}} \sigma_1 \xrightarrow{t_1} ...$ defines the sequence of length $len(\pi)$ where $\sigma_i = (l_i, v_i)$ is a state with a location l_i and valuation v_i, $t_i \in \mathbb{R}_{\geq 0}$ is the duration of a flow. π is an initial path if $\sigma_0 = (\ell_0, \nu_0)$ is an initial state, which is the case if $\nu_0 \in Init(\ell_0)$. σ is reachable only if there is an initial path leading to σ [WRÁ24].

2.2 Linear Hybrid Automata

Hybrid automata can be subdivided into several subclasses. Each subclass is restricting how the predicates occurring in flows, invariants, guards, and resets can be defined [Ábr21]. The subclass which was possible to parse in RealySt before was mostly the rectangular automaton with random clocks. Here, all assertion predicates are restricted to rectangular sets [DSÁR23]. Since linear terms over variables are not possible but can provide a more precise representation, the subclass of linear hybrid automata (LHA) is introduced [Sch19].
A linear term is a linear combination of variables in Var with rational coefficients [Ábr21]. An affine term extends this notion by adding a constant offset to the linear combination [Sch19]. In the following, we treat linear and affine terms interchangeably, since our extension for the parser does not differentiate between linear and affine terms and handles them equally.

2.2.1 Linear Hybrid Automata I

Linear hybrid automata can be subdivided into two classes. An LHA I is a time-deterministic hybrid automaton that contains linear and affine terms.
Although the invariants, guards and resets can consist of linear terms, the flows are still defined by rectangular sets [Sch19].

2.2.2 Linear Hybrid Automata II

In this work, we want to focus on the second type of LHA because the extension of our work mainly focused parsing LHA II. They are similar to LHA I except that the flows are defined by linear terms [Ábr21]. We have based the following definition on

the previous definition 2.1 of hybrid automata and RealySt to ensure that the tool's internal representation and the definition are consistent.
We denote $\mathbb{I}$ as the set of all intervals $[a,b], [a,\infty), (-\infty,b], (-\infty,\infty) \subseteq \mathbb{R}$ with rational endpoints a, $b \in \mathbb{Q}$, interpreted with the standard set semantics [DSÁR23].
The main difference in comparison to the first type is the structure of the flows in a location. Here, they are never defined as rectangular sets but with linear ordinary differential equations $\dot{x} = A * x + b$. The other constraints can be either rectangular or linear now [Ábr21]. A is a matrix and b a vector, according to the definition in [Sch19].

Definition 2.2.2.1 (Linear Hybrid Automata II Syntax). A linear hybrid automaton II is a tuple $H = (Loc, Var, Flow, Inv, Lab, Edge, Init)$ where

- Loc is a non-empty and finite *set of locations*.
- $Var = \{x_1, ..., x_d\}$ is a finite ordered *set of variables*.
- $Flow : Loc \to (\mathbb{R}^d \to \mathbb{R}^d)$ defines the *time-progress* in a location. The flow of an arbitrary variable $x \in Var$ is $\dot{x_i} = A_{loc}(i) * x + b_{loc}(i)$ where $A_{loc} \in \mathbb{R}^{n \times n}$ and $b_{loc} \in \mathbb{R}^n$. $A_{loc}(i)$ is the i-th row of A_{loc} and $b_{loc}(i)$ the i-th component of b_{loc}. If the flow is affine, it depends on a linear term and a constant, meaning that $A_{loc}(i) \neq 0$ and $b_{loc}(i) \neq 0$. If the flow is only linear and not affine, then $b_{loc}(i) = 0$. If it is only a constant flow, then $A_{loc}(i) = 0$.
- $Inv : Loc \to 2^{\mathbb{R}^d}$ specifies an *invariant* for each location, represented as a linear set (polytope) of the form $\{x \in \mathbb{R}^d \mid Ax \leq b\}$, where $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$.
- $Lab = \{a_1,...,a_k\}$ is a non-empty finite ordered *set of labels*.
- $Edge \subseteq Loc \times Lab \times \mathcal{G} \times \mathcal{R} \times Loc$ is a finite set of discrete *transitions or jumps*. For a jump $(\ell, a, g, r, \ell') \in Edge$, ℓ and ℓ' are its source respectively target locations, a is its label, g its *guard*, and r its *reset*.
- $Init : Loc \to 2^{\mathbb{R}^d}$ defines *initial valuations* for each location and linear start conditions for Var. We call a state $(\ell, \nu) \in \Sigma$ initial if $\nu \in Inv(\ell) \cap Init(\ell)$.

Here, $\mathcal{G}$ denotes the set of admissible guard constraints, defined as $\mathcal{G} := \{x \in \mathbb{R}^d \mid Ax \leq b\} \cup \mathbb{I}^d$, where $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$. Guards are either given as linear constraints of the form $Ax \leq b$ or as rectangular constraints, i.e. conjunctions of interval constraints of the form $a \leq x_i \leq b$.
Similarly, $\mathcal{R}$ denotes the set of admissible reset relations, defined as $\mathcal{R} := \{x' = Rx + r \mid R \in \mathbb{R}^{d \times d}, r \in \mathbb{R}^d\} \cup \mathbb{I}^d$, where a reset is either given as an affine-linear assignment of the form $x' = Rx + r$, or as a rectangular constraint, i.e. $x'_i \in [a_i,b_i]$.

Example 2.2.2.2 (Linear Hybrid Automata II Heat).

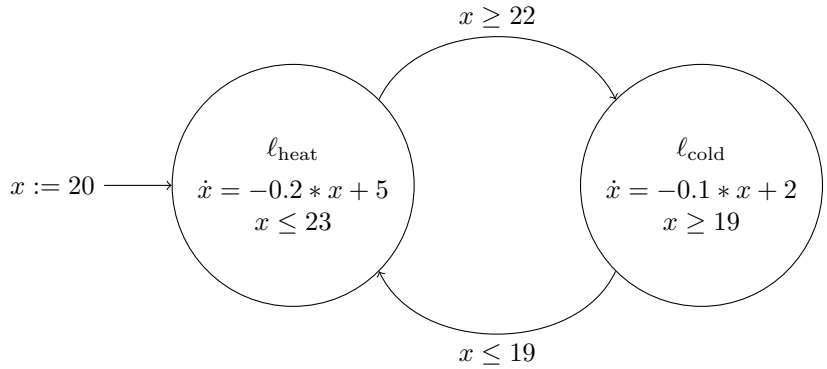

Figure 2.2: Example Heating System as a LHA II

This example extends the previous one in Figure 2.1 by illustrating how linear flows can be incorporated into a hybrid automaton. In contrast to the earlier automaton, which only used constant rates, this example introduces flows that depend linearly on the current state, thereby resulting in a linear hybrid automaton of type LHA II. The purpose of this example is to demonstrate how hybrid automata can be extended with linear dynamics, allowing the continuous evolution of variables to depend not only on constants but also on other variables. In particular, the derivative of x is now defined by a linear expression, meaning that the value of x increases or decreases according to this function. This highlights how more expressive system behaviour can be modelled by extending basic hybrid automata with linear flows.

Definition 2.2.2.3 (Linear Hybrid Automata II with Random Clocks). LHA II can be extended by random clocks to model stochastic behaviour and capture uncertainty in the timing of transitions [HHHK13]. In particular, they allow the representation of nondeterministic choices that depend on randomly sampled delays. The following definition is based on [BHR24] and the definition of linear hybrid automata II in 2.2.2.
A Linear Hybrid Automaton II with Random Clocks $LHARC\ II$ is a tuple
$H = (Loc,\ Var,\ Flow, Inv,\ Lab,\ Edge,\ Init,\ Dist,\ Stoch)$ where

- Loc is a non-empty and finite *set of locations.*
- $Var = Var_{con} \cup Var_r = \{x_1, ..., x_d\}$ is a finite ordered *set of continuous* (Var_{con}) and *random clock* (Var_r) *variables.*
- $Flow : Loc \to (\mathbb{R}^d \to \mathbb{R}^d)$ defines the *time-progress* in a location. The flow of an arbitrary variable $x \in Var_{con}$ is $\dot{x_i} = A_{loc}(i) * x + b_{loc}(i)$ where $A_{loc} \in \mathbb{R}^{n \times n}$ and $b_{loc} \in \mathbb{R}^n$. $A_{loc}(i)$ is the i-th row of A_{loc} and $b_{loc}(i)$ the i-th component of b_{loc}. If the flow is affine, it depends on a linear term and a constant, meaning that $A_{loc}(i) \neq 0$ and $b_{loc}(i) \neq 0$. If the flow is only linear and not affine, then $b_{loc}(i) = 0$. If it is only a constant flow, then $A_{loc}(i) = 0$.
  The flow for all $x \in Var_r$ can be either -1 or 0.
- $Inv : Loc \to 2^{\mathbb{R}^d}$ specifies an *invariant* for each location, represented as a linear set (polytope) of the form $\{x \in \mathbb{R}^d \mid Ax \leq b\}$, where $A \in \mathbb{R}^{n \times d}$ and $b \in \mathbb{R}^n$.

- $Lab = \{a_1,...,a_k\}$ is a non-empty finite *ordered set of labels.*
- $Edge \subseteq Loc \times Lab \times \mathcal{G} \times \mathcal{R} \times Loc$ is a finite set of discrete *transitions or jumps.* For a jump $(\ell, a, g, r, \ell') \in Edge$, ℓ and ℓ' are its source respectively target locations, a is its label, g its *guard*, and r its *reset.* Guards must be pairwise disjoint for each pair of jumps with identical source location and label.
- $Init : Loc \to 2^{\mathbb{R}^d}$ defines *initial valuations* for each location and linear start conditions for Var. We call a state $(\ell, \nu) \in \Sigma$ initial if $\nu \in Inv(\ell) \cap Init(\ell)$.
- $Dist : Var_r \to P(\mathbb{R}_{\geq 0})$ gives every random clock a *probability distribution.*
- $Stoch \subseteq Edge \times Var_r$ is a set of edges containing *stochastic transitions* which are activated by a random clock. Activated means a random clock $r \in Var_r$ reached $r = 0$.

Here, $\mathcal{G}$ denotes the set of admissible guard constraints, defined as $\mathcal{G} := \{x \in \mathbb{R}^d \mid Ax \leq b\} \cup \mathbb{I}^d$, where $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^m$. Guards are either given as linear constraints of the form $Ax \leq b$ or as rectangular constraints, i.e. conjunctions of interval constraints of the form $x_i \in [a_i,b_i]$.
Similarly, $\mathcal{R}$ denotes the set of admissible reset relations, defined as $\mathcal{R} := \{x' = Rx + r \mid R \in \mathbb{R}^{d \times d},\ r \in \mathbb{R}^d\} \cup \mathbb{I}^d$, where a reset is either given as an affine-linear assignment of the form $x' = Rx + r$, or as a rectangular constraint, i.e. $x'_i \in [a_i,b_i]$.
This definition reflects the extensions implemented in the parser, which supports both rectangular and linear guards as well as affine resets. Now LHA II can be extended with random events by introducing random clock variables. The expiration times are sampled from continuous probability distributions.

Example 2.2.2.4 (Linear Hybrid Automata II Heat with a Random Clock).

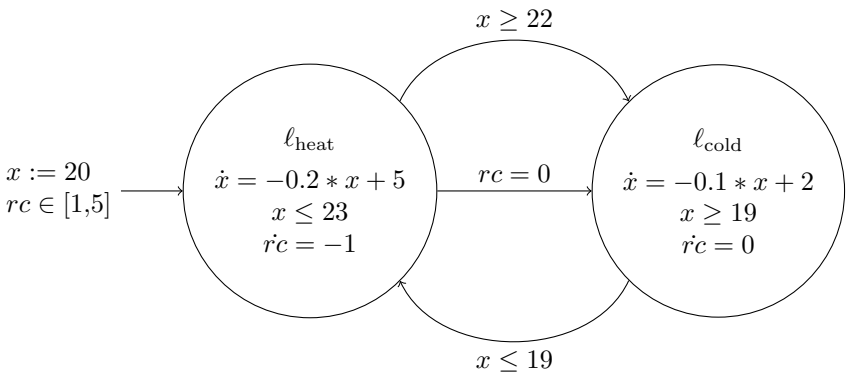

Figure 2.3: Example Heating System as a LHA II with Random Clocks

This example extends the previous linear hybrid automaton in Figure 2.2 by introducing a random clock rc. It illustrates how a deterministic linear hybrid automaton can be extended to include stochastic behaviour, resulting in a stochastic hybrid automaton. In particular, a new stochastic transition is added between the locations, and the random clock rc is assigned a random value between one and five during the

initialization. The transition is triggered once rc reaches zero. This demonstrates how random clocks can be used to model nondeterministic timing behaviour.

Definition 2.2.2.5 (LHARC II Semantics). The definition of the semantics is inspired by the semantics of rectangular automaton with random clocks in [DSÁR23]. Let H be a LHARC II. The state $\sigma = (l, v, \mu, s)$ with a location $l \in Loc$, values of the continuous variables $v \in \mathbb{R}^d$, values of the stochastic variables $\mu \in \mathbb{R}^{d_R}_{\geq 0}$ and expiration times $R \in \mathbb{R}^{d_R}_{\geq 0}$. The evolution of the state of H can be specified in the following:

$$\frac{\begin{array}{l} t \in \mathbb{R}_{\geq 0},\ \dot{v} \in Flow(\ell) \\ v' = f(t)\ \ v' \in Inv(\ell) \\ \forall r \in Var_r :\ \mu'_r = \begin{cases} \mu_r - t & \text{if } r \text{ is enabled} \\ \mu_r & \text{otherwise} \end{cases},\ 0 \leq \mu' \end{array}}{(\ell, v, \mu, R) \xrightarrow{t} (\ell, v', \mu', R)} \quad \textsc{Flow}$$

$$\frac{\begin{array}{l} e = (\ell, a, g, reset, \ell') \in Edge \\ e \notin Stoch \qquad v \in g \\ (v,v') \in reset \qquad v' \in Inv(\ell') \end{array}}{(\ell, v, \mu, R) \xrightarrow{e} (\ell', v', \mu, R)} \quad \textsc{Jump}_{con}$$

$$\frac{\begin{array}{l} e = (\ell, a, g, reset, \ell') \in Edge,\ (e,r) \in Stoch \\ v \in g,\ \mu_r = 0,\ (v,v') \in reset,\ v' \in Inv(\ell') \\ R'_r \sim Dist(r),\ \mu'_r = R'_r \\ \forall r' \neq r :\ \mu'_{r'} = \mu_{r'},\ R'_{r'} = R_{r'} \end{array}}{(\ell, v, \mu, R) \xrightarrow{e} (\ell', v', \mu', R')} \quad \textsc{Jump}_{r}$$

A path is initial if $v_0 \models Init_H(l_0)$. If an initial path leads to a state σ, then the state is reachable [Sch19].

2.3 Reachability Analysis

A central problem in the verification of hybrid automata is to decide if there exists an execution from a given initial location to a given goal location or state. The calculation of the probability of this execution can be done by reachability analysis. This problem is undecidable for linear hybrid automata, but decidable for rectangular automata [BDG+11].
The reachability analysis is performed by iteratively computing the set of reachable states. The algorithm starts with an initial state set and repeatedly computes the successor states obtained by either continuous evolution of the variables or discrete transitions between the locations. These new states are added to the set of reachable states. From there, the algorithm evolves over time, adding the new states to the set of states reached so far, as well as the transitions. This process is repeated until no new states are discovered, thereby reaching a fixed point [LG09].

Since the reachability problem is undecidable for many classes of hybrid automata, the exact computation of reachable sets is often infeasible. Therefore, the reachability analysis relies on approximations or restrictions of the model [LG09]. One approach is to consider bounded reachability, where the analysis is restricted to a finite time horizon T and a fixed jump depth. This can be achieved by introducing a new additional clock variable t with its derivative $\dot{t} = 1$ and restricting the automata to states where $t \leq T$ [LG09]. Another possible problem is a *Zeno* execution which can rise, meaning that an infinite number of transitions in a finite amount of time can be taken. An example is a bouncing ball, where the first impact occurs after one second, the second after half a second, and so on. In a hybrid automaton model that disregards physical effects such as air resistance or material deformation, the simulation of a bouncing ball with this automaton would lead to infinitely many impacts occurring in a finite amount of time, as the ball continuously loses energy and reaches ever smaller heights [LG09]. To avoid the *Zeno* behaviour, the jump depth of discrete transitions is also limited. Under such restrictions, reachability analysis can be effectively performed using iterative fixed-point computations. The result is an approximation of the reachable state space, which can then be used to verify whether certain goal states are reachable or whether unsafe states can be avoided [LG09].

Chapter 3

Tools

This chapter introduces the tools used in this work. In particular, we present RealySt [AG 25], a framework which is used for modelling and analysing stochastic hybrid automata, and HyPro[hyp25], which provides the underlying data structures and algorithms for reachability analysis. Both tools play a central role in the implementation and evaluation of the parser extensions described in this thesis.

3.1 HyPro

HyPro is an open-source C++ library that provides implementations of state set representations for the reachability analysis of hybrid automata via flowpipe construction. It supports several representations like boxes, convex polytopes, support functions, and zonotopes. All implementations conform to a unified interface allowing users to seamlessly switch between representations, even during the analysis [hyp25]. The computed sets correspond to over-approximations of the reachable states of the system. They can be used for safety verification by checking whether the reachable states intersect with a predefined set of unsafe sets. During analysis with flowpipe-construction, the algorithm computes the sets iteratively, similar to the reachability analysis described previously in 2.3 [SÁMK17].
HyPro provides different geometric abstractions to represent these sets. For example, polytopes are represented as the intersection of half-spaces defined by a matrix A and a vector b, i.e., $P = \{x \mid A \cdot x \leq b\}$ [HSRÁ17].
In addition to these representations, the library offers the core operations required for reachability analysis of linear hybrid automata, including linear transformations, Minkowski sums, intersections, unions, and emptiness checks. It further supports conversions between different state set representations, either exactly or via over-approximation, allowing a flexible trade-off between precision and efficiency. To control the complexity of these representations, the library includes several reduction techniques, such as number reduction for boxes and polytopes, order reduction for zonotopes, and simplification of support function representations [SÁMK17].
Besides its core functionality, HyPro offers auxiliary features such as plotting via `gnuplot` or `TikZ`, logging mechanisms, and example implementations of reachability algorithms. It also provides benchmark examples for evaluating the performance of different state set representations in practice [SÁMK17].

3.2 RealySt

The tool RealySt, introduced in [DSSR23], is used to compute the reachability of different hybrid automata. RealySt is an open-source tool written in C++ to compute the maximum reachability probability of singular automata with random clocks and rectangular automata with random clocks. Geometric operations on convex polytopes in RealySt are performed using the library HyPro [hyp25], which also computes the reachable state space as a flowpipe [SÁMK17].

3.2.1 Procedure of RealySt

RealySt is a command-line tool for Linux and the Windows Subsystem for Linux. It can be invoked as `./realyst` and with a list of parameters [AG 25]:

- `-t [number]` sets the global time bound to the reachability analysis to the given number.
- `-d [number]` sets the maximal jump depth.
- `-i [number]` defines the number of integration samples for Monte-Carlo.
- `-b [BENCHMARK]` selects the benchmark where the automaton is defined.
- `-m [CASE]` chooses the case of the benchmark to select the automaton.
- `--from-jani [FILEPATH]` parses an automaton from a Jani-file.
- `-l [info/trace/debug]` shows the log information.
- `--plotDimensions [index1] [index2]` prints the results in a coordinate system where the variable with index1 is on the x-axis and the variable with index2 on the y-axis.
- `--LTIAnalysis` if it is a linear hybrid automaton, this flag has to be set, otherwise, it expects a rectangular one.
- `--createJani [FILENAME]` creates a Jani-file with the automaton in a new file with the given name.

During the execution of one command line, RealySt outputs everything depending on the chosen parameter with the logging level and the reachability probability.

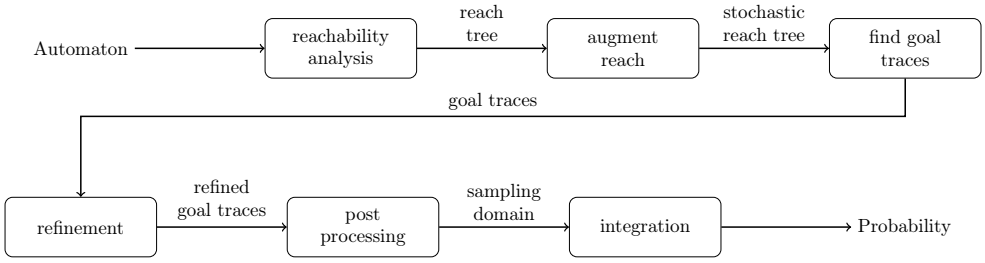

Figure 3.1: Process Flow of RealySt [DSSR23].

After being invoked through the command line, RealySt proceeds with the core program flow for probabilistic reachability analysis, which is shown in Figure 3.1. The explanation of the work flow is from [DSSR23]. The tool is given a time bound t_{max} and a goal specification in order to perform the reachability analysis in several stages. The analysis approach implemented in RealySt combines classical reachability techniques with stochastic evaluation. Firstly, a standard reachability analysis is performed to produce a reach tree [Sch19], representing the set of reachable states using convex polytopes. During this phase, random clocks are treated as continuous variables to capture all possible values. Based on this reach tree, stochastic information is incorporated by associating random delays with specific clock variables. This enriches the model with probabilistic behaviour, allowing both nondeterministic and stochastic aspects of the system to be represented. Using the augmented reach tree, paths leading to the goal are identified. For each node whose state set intersects the goal region, a corresponding trace is constructed from the root to that node. The associated state sets are then restricted to the goal region, yielding subsets relevant for further analysis. A backward refinement step is then applied to determine which parts of the state space can reach the goal. Starting from the refined goal segments, the reachable sets are partitioned according to their ability to lead to successful execution. Constraints on the random clocks are then derived for each resulting trace, describing the conditions under which the system follows that path. Finally, the reachability probability is approximated by integrating over these regions. Monte Carlo methods, such as the VEGAS algorithm, are used to achieve this, providing estimates of the optimal reachability probability alongside statistical and numerical error bounds.

3.2.2 Model Representation in RealySt

RealySt can either parse a hybrid automaton using a given Jani-file or define the automaton from code. In both cases, how the automaton attributes are presented in RealySt is important, since it directly influences how the model is interpreted and analysed within the tool, as well as how it can be translated back into a Jani-file.
In RealySt, hybrid automata are constructed using the C++ interface from HyPro. This interface is used to define variables, locations, flows, invariants, transitions and specifications. The framework provides a structured, modular approach that enables rectangular and linear hybrid automata to be represented. The following code snippets are created based on the benchmarks in [AG 25], which contains examples of automata defined in the internal representation.

Variables. In RealySt, two types of variables can be created. Continuous variables are typically used to represent physical quantities, such as temperature, while stochastic variables model random delays or probabilistic behaviour. Variables are created using:

```
poolPtr->createVariable("variable_name",
 ↪  librealyst::variableType::CONTINUOUS);
poolPtr->createVariable("variable_name",
 ↪  librealyst::variableType::STOCHASTIC);
```

Locations. The discrete behaviour of a hybrid automaton is defined by its locations. Each location represents a system operation mode and is created using the command:

```
automaton.createLocation("location_name");
```

Flows. For each location, the continuous behaviour of the automaton is specified via two types of flows. RealySt supports rectangular flows, where the derivative of each variable is bounded by an interval. These flows are defined using `VariableIntervalMap`, where each variable index is associated with an interval:

```
VariableIntervalMap flow_name {
    {indexOfVariable, carl::Interval<Number>{lower,
    ↪ upper}}
};
automaton.setRectangularFlowInLocation(location_name,
↪ flow_name);
```

In addition to rectangular modelling, RealySt also supports linear hybrid automata with affine dynamics. In this case, flows are not specified by interval bounds on derivatives, but by linear flow matrices that describe systems of ordinary differential equations of the form $\dot{x} = Ax + b$. In the implementation, such flows are represented using `hypro::linearFlow<Number>` together with a flow matrix, which is then assigned to a location via:

```
automaton.setLinearFlowInLocation(location_name, flow);
```

This makes it possible to express dependencies between variables explicitly. In the battery lifetime model [RHH25], for example, the evolution of one variable depends on the values of several others.

Invariants. Invariants restrict the admissible values of variables within a location. In the rectangular case, they are defined using intervals:

```
VariableIntervalMap invariant{
    {indexOfX, carl::Interval<Number>{lower_bound,
    ↪ upper_bound}}
};
automaton.setInvariantsInLocation(loc_idle, invariant);
```

In the linear case, invariants can be expressed as general constraints $Ax \leq b$.

```
hypro::matrix_t<Number> I =
↪ hypro::matrix_t<Number>::Zero(1,2);
hypro::vector_t<Number> i(1);

I(0, indexOfX) = 1;
i(0) = 10;

hypro::ConstraintSet<Number> invSet(I, i);
automaton.setInvariant(loc, invSet);
```

Transitions and Guards. Transitions define the discrete evolution between locations. They may optionally be associated with a stochastic variable:

```
auto* transition = automaton.createTransition(loc_idle,
↪ loc_run, indexOfrc, true);
```

Guards determine when a transition is enabled. Rectangular guards use interval constraints:

```
VariableIntervalMap guard{
    {indexOfX, carl::Interval<Number>{lower_bound,
     ↪  carl::BoundType::WEAK,upper_bound,
     ↪  carl::BoundType::INFTY}}
};
automaton.setGuardsOfTransition(transition, guard);
```

In the linear case, guards can also be defined as general linear inequalities $Ax \leq b$:

```
hypro::matrix_t<Number> G =
 ↪  hypro::matrix_t<Number>::Zero(1,2);
hypro::vector_t<Number> g(1);

G(0, indexOfX) = -1;
g(0) = -5;
```

Resets. Resets define how variables are updated when a transition is taken. In the general case, RealySt supports affine transformations $x' = Rx + r$:

```
hypro::matrix_t<Number> R =
 ↪  hypro::matrix_t<Number>::Identity(2,2);
hypro::vector_t<Number> r =
 ↪  hypro::vector_t<Number>::Zero(2);

R(indexOfX, indexOfX) = 0;
r(indexOfX) = 5.0;

hypro::AffineTransformation<Number> reset(R, r);
automaton.setAffineResetsOfTransition(transition, reset);
```

This allows both constant resets and linear transformations of the state.

Initial States. The initial state set is defined using constraints over the variables:

```
VariableIntervalMap initial{
    {indexOfX, carl::Interval<Number>{0.0}},
    {indexOfrc, carl::Interval<Number>{0.0}}
};
automaton.setInitialStates(loc_idle, initial);
```

Specifications. Goal specifications are defined as constraint sets over the variables. Internally, these are represented using matrix-vector representations of linear inequalities of the form $Ax \leq b$:

```
hypro::matrix_t<Number> specMatrix =
 ↪  hypro::matrix_t<Number>::Zero(n, d);
hypro::vector_t<Number> specVector(n);
```

```
hypro::ConstraintSet<Number> constraints(specMatrix,
 ↪  specVector);
librealyst::Specification specification{};
specification.constraints =
 ↪  hypro::Condition<Number>(constraints);
spec.addSpecification(specification);
```

Matrix. For defining linear terms in RealySt, matrices are used. An example is given by the following code snippet:

```
hypro::matrix_t<Number> A =
 ↪  hypro::matrix_t<Number>::Zero(3,3);
A(0, 0) = 2;
A(0, 1) = 4;
A(0, 2) = 1;

A(1, 1) = 5;
A(1, 2) = 3;
```

Here, a matrix $A \in \mathbb{R}^{3\times 3}$ is initialized with all its entries set to zero, and then filled to encode the linear term. The resulting matrix is:

$$\begin{pmatrix} 2 & 4 & 1 \\ 0 & 5 & 3 \\ 0 & 0 & 0 \end{pmatrix}.$$

This matrix can be used to define the flows of two variables x and y in a location, where the index of x is 0 and the index of y is 1. Each row of the matrix corresponds to the derivative of one variable. Hence, the first row defines the flow of x, while the second row defines the flow of y. The columns represent the influence of the variables on the derivatives. The first column corresponds to the coefficient of x, the second column to the coefficient of y, and the third column represents the constant part of the expression. Consequently, the matrix encodes the system of differential equations $\dot{x} = 2x + 4y + 1$ and $\dot{y} = 5y + 3$.

Chapter 4

Jani

The JANI specification provides a unified and tool-independent format for modelling and analysing stochastic hybrid systems [BDH+17]. In this chapter, we introduce the main concepts of JANI, describe the structure of JANI models, and discuss the current state of the parser used to translate between JANI and REALYST.

4.1 Origin of Jani

Several other tools besides REALYST analyse the probability of hybrid automata. However, the internal representations of automata from different tools are not always compatible, meaning they could not be exchanged. Therefore, the `jani-specification` was introduced as an exchange format [Zum24]. JANI consists of the `jani-model`, which allows the creation of stochastic hybrid automata, and `jani-interaction`, which should provide a stable interface for reusing existing implementations. Although the latter is outdated, it is briefly described for completeness, as it reflects the original design of the JANI framework and its intended integration of analysis tools [Zum24].
JANI uses the JSON data format to encode its models and messages. JSON itself is a language-independent format that represents data in the form of objects, arrays and primitive values [BDH+17]. The structure of valid JANI models, as well as the encoding of interaction messages, is specified using `js-schema`. This lightweight schema language allows the definition and validation of object structures, but it cannot express certain complex constraints such as XOR relations between attributes [BDH+17].
The `jani-model` defines the structure of models and provides a set of variable types, expressions, and assignments, together with commonly used operations. It also allows the specification of probabilistic properties, which can be used to describe and analyse quantitative aspects of a system [BDH+17]. The format is designed to be simple, so that it can be implemented and used by different tools. At the same time, the use of JSON makes it possible to extend the format without breaking existing models [BDH+17].
The second component, `jani-interaction`, specifies a communication protocol between tools. Following a client-server architecture, it aims to provide a stable interface for integrating and reusing existing verification implementations [BDH+17]. This approach reduces setup complexity and supports the execution of tools in a

distributed manner, as well as facilitates the development of unified graphical user interfaces for JANI-based tools [BDH+17]. The protocol is based on the concept of roles, where the currently defined analyse role provides access to verification procedures [BDH+17]. Tools like Modest or Prism, that implement this role, offer a set of analysis engines, each of which corresponds to a specific verification algorithm [BDH+17]. The role-based design also allows for future extensions, such as additional roles for model transformation or conversion [BDH+17].

4.2 Structure of a Jani-File

A JANI model is defined as a structured JSON object that describes hybrid automata in a tool-independent format. It contains several basic attributes, including the model name, its type, the version of the `jani-model` specification, a list of global variables, optional initial restrictions, and a description of the automata's composition [Zum24]. The following explanations, how different attributes of hybrid automata are represented in JANI, are from [Zum24]:

- The **variables** have the attribute `name`, saved as a string, and a `type`. Even though a variable can be either `STOCHASTIC` or `CONTINUOUS`, the `jani-model` does not allow `STOCHASTIC`, so they are all saved as `CONTINUOUS`. The core behaviour of the system is defined within automata, which consist of discrete locations and transitions.
- Each **location** represents a discrete state of the system and is associated with continuous dynamics. These dynamics, together with invariants, are specified using the `time-progress` attribute. Growth rates of variables can be expressed using the derivative operator `der`.
- **Invariants** restrict the staying within a location depending on the variables' values. In the JANI-file, these constraints are defined in a location.
- **Transitions** describe the discrete evolution between locations. Each transition consists of a single source location, a (possibly non-empty) set of target locations, a guard condition, and a list of resets associated with each target.
- **Guards** are logical expressions that determine whether a transition may be taken. In contrast to invariants, which are part of the `time-progress` condition of a location, guards in JANI are stored explicitly as guard expressions within transitions.
- **Stochastic behaviour** in JANI is typically modelled through variables that are assigned values according to probability distributions. These assignments occur as part of transitions. A common pattern is to use additional conditions in guards to ensure that transitions depending on stochastic variables are only taken when these variables reach specific values.
- The **specification** of properties in JANI allows the formulation of verification goals directly within the model. These goals are typically expressed as properties over system states or execution paths. A specification may refer to discrete locations, variable values, or a combination of both. Conditions over variables are expressed using logical expressions, similar to guards and invariants. These

expressions are restricted to comparisons with constants and simple arithmetic operations.

Jani provides constructs such as filter operators to evaluate properties over sets of states, for example, to determine whether a certain condition holds in all or some reachable states. In this way, specifications can be used to define goal conditions and to verify whether they are satisfied during analysis [Zum24].

4.3 Current State of the Parser between Jani and RealySt

The current state of the parser in RealySt was described in [Zum24]. Before the introduction of our extension, the Jani parser in RealySt, which converts a Jani-file into the tool's internal representation, offered limited functionality and supported only a subset of the Jani specification. Regarding variable types, the parser supported only a limited set of basic types, namely `bool`, `int`, and `real`, as well as bounded types and continuous variables. Variables defined in Jani could be either global or local to an automaton. However, due to the single-automaton restriction, all variables are treated as global during parsing in RealySt. Stochastic behaviour is only partially supported. Since Jani variables do not include attributes for probability distributions, stochasticity cannot be directly associated with variables. Instead, probability distributions were handled within assignments of transitions. To initialise stochastic variables, the parser introduced an additional discrete initial location. This location is not part of the original model semantics but served as a technical construct. It is included in the set of initial states, had no incoming transitions, and its outgoing transitions contained only assignments for stochastic variables, typically without guards or with trivially true guards. The assignments were analysed to detect probability distributions, which enabled the parser to identify stochastic behaviour in transitions. Nevertheless, this mechanism was limited and did not fully capture all stochastic modelling aspects of Jani. Further limitations concerned the supported modelling features. The parser was only capable of handling rectangular automata, meaning that constraints were restricted to interval bounds and comparisons with constants. More general linear expressions were not supported at this stage. In addition, resets could not be parsed or generated, which further restricted the expressiveness of transitions. Finally, specifications could be parsed from Jani to RealySt, but were missing while creating a Jani-file from RealySt. While RealySt itself allows the definition of goal conditions, such as sets of goal locations or constraints over variable values, these could not be parsed from or exported to Jani models. As a result, verification properties defined in Jani were not available within the parser.
Overall, the initial version of the parser provided a basic translation mechanism for simple Jani models but lacked support for several important features, including networks of automata, linear hybrid automata, resets, and specifications. These limitations motivated the extensions developed in the course of this work.
In summary, although Jani provides a flexible modelling format, the existing parser in RealySt was not able to handle linear hybrid automata. In particular, linear flows, linear guards, and linear invariants were not supported, as the parser was restricted to rectangular constraints. Furthermore, resets were not handled at all, and specifications could not be created when exporting models to Jani. In this work, we extend the parser to support linear hybrid automata by enabling the parsing of linear

expressions in flows, guards, and invariants. In addition, we introduce support for affine resets of the form $x := Ax + b$. We also extend the handling of specifications for the export. These extensions are implemented and described in detail in the next chapter.

Chapter 5

Implementation

This chapter explains how we extended the JANI parser in REALYST to enable it to parse linear hybrid automata. While working on this, we also identified some missing functionalities such as parsing resets in general, and handling specifications while creating a JANI-file. We incorporated these into the parser.

5.1 Parsing Linear Hybrid Automata

5.1.1 Linear Expression in JANI

Before we started working on the parser, we planned the structure of a linear expression in a JANI-file. The initial idea was to define linear terms as an array $[[a_1, x_1,],...[a_n, x_n], c]$ since this format would be easier to edit by hand. However, we realised this idea would have required us to change the file `jani-schema.json`, meaning that the JANI-files would no longer be compatible with other tools such as Modest [mod]. For this reason, we opted for a nested tree structure for linear terms, as shown in the example in Figure 5.1 where the term $2 * x + 3 * y + 5$ is defined.

We now expect an abstract syntax tree [ZWZ+19] for linear attributes in which the first node splits with the operators "$=$", "$\leq$" or "*der*", the middle nodes are connected with "$+$" or "$-$", and the end nodes, which are pairs of variables and coefficients, with "$*$". The "$*$" operator can also occur to connect a number with a linear term. But it can never occur between variables to ensure linearity. Conversely, when exporting to JANI, the internal representation of REALYST is reconstructed into an abstract syntax tree using nested addition and constant–variable multiplications.

5.1.2 Adding Linear Representation

Several new data structures and type definitions were introduced to support linear hybrid automata. In particular, custom types such as `LinearExpression` and `LinearFlow` are used to represent linear terms. A `LinearExpression` stores a linear combination of variables together with a constant term, by mapping variables to their coefficients with their index and a separate constant value. Based on this, a `LinearFlow` represents the dynamics of a system by assigning such linear expressions to the derivatives of variables, corresponding to equations of the form $\dot{x} = Ax + b$. Mappings like `GeneralFlowMap` provide a unified representation of flows, guards,

invariants, and properties, enabling both interval-based and linear constraints to be handled via a variant type. This allows the parser to process both types of models in a consistent way. Additionally, a global configuration flag is introduced to control whether the parameter `"-LTIAnalysis"` is set.

```
{
  "op": "$+$",
  "left": {
    "op": "$+$",
    "left": {
      "op": "$*$",
      "left": 2,
      "right": "x"
    },
    "right": {
      "op": "$*$",
      "left": 3,
      "right": "y"
    }
  },
  "right": 5
}
```

Figure 5.1: Example of a linear expression in JANI format

5.1.3 Parsing of Linear Expressions

After we understood how a linear expression is structured in JANI, we added two functions in `JaniToRealyst.tpp`. They can be used for all attributes of an automaton that can be defined as a linear expression since the linear terms are structured the same way in the file.
The first function is `isLinearExpression(const json& exp)` which gets an expression of the JANI-file and checks whether the expression is linear. This is necessary since there is no specific operator that can be used immediately to identify whether the expression is a linear term. Therefore, we determine whether the given expression is nested with "+", "−" or "∗" in the outer parts. If it is "+" or "−", we check whether the nested terms on the left-hand and right-hand side of these operators are linear expressions. When a "∗" is detected, we check if one side is a number and the other side is linear. The functions calls itself recursively until it detects an invalid input, which would lead to `false`, or finds only a number or variable, leading to `true`. If all created subexpressions satisfy the linearity conditions, the function returns `true` for the entire expression. The flow of the function is also presented in the Figure 5.2.
To illustrate how the function works, we consider as an example the linear expression $2 * x + 3 * y + 5$ from the JANI expression in Figure 5.1. The function first identifies the outermost "+" operator and splits the expression into the left term $2 * x + 3 * y$ and the right term 5. The function does not always split the constant part from the rest of the term at first. The order and which "+" is detected first depends on the

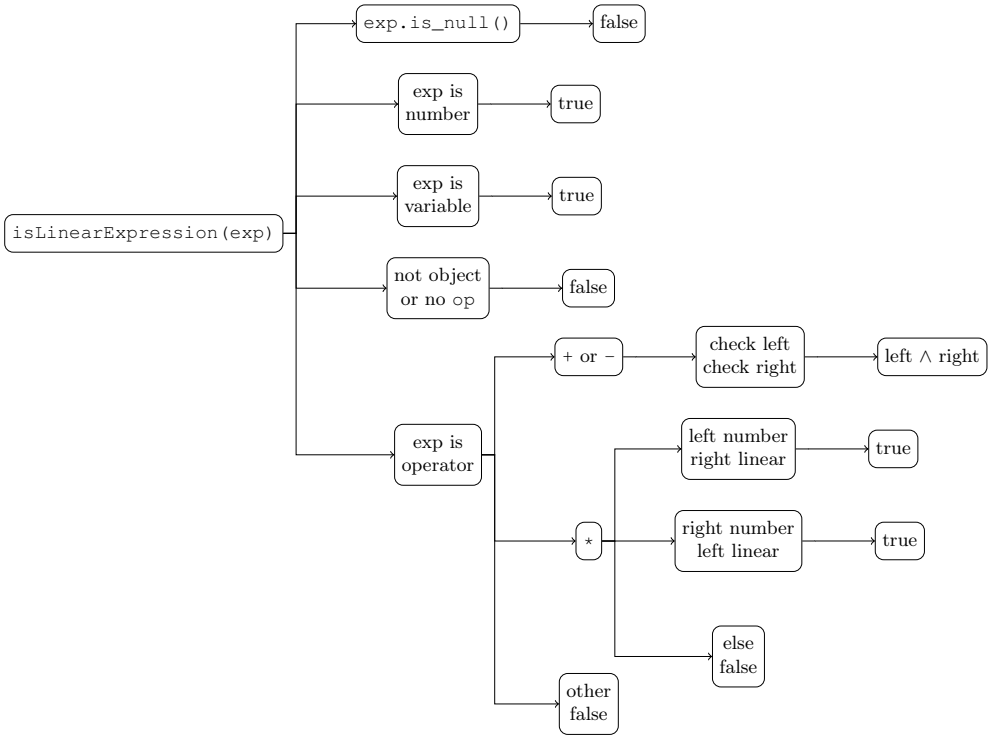

Figure 5.2: Decision tree for checking if an expression is linear.

structure of the JANI-file . Both subexpressions are then checked recursively to determine whether they are linear. When `isLinearExpression(...)` is called with $2 * x + 3 * y$, the function again detects a "+" operator and splits the expression into $2*x$ and $3*y$. For the term $2*x$, the function identifies the "$*$" operator and checks if one operand is a number and the other is a variable. Since 2 is a number, the function recursively calls itself on the right operand x. As x is a variable, this recursive call returns `true`, and therefore the subexpression $2 * x$ is classified as linear. The same reasoning applies to $3 * y$, and the constant term 5 is also considered linear. Since all subexpressions satisfy the linearity conditions, the function returns `true` for the entire expression.

In addition to a function for verifying linear expressions, we have also added a function to parse linear expressions of all possible attributes:

```
parseLinearExpression(const json& exp, WeightedVariables&
↪  weights, Number& constant, Number factor)
```

The function transforms a JANI expression into an internal linear representation. Its purpose is to extract the coefficients of the variables and the constant part of an expression, storing them separately. This separation is required by the internal representation used in REALYST, where linear expressions are handled as a combination of weighted variables and a constant term. Therefore, the parser must transform JANI expressions into this format. The flow of `parseLinearExpression(...)` is also presented in the Figure 5.3. The function operates recursively on the abstract syntax

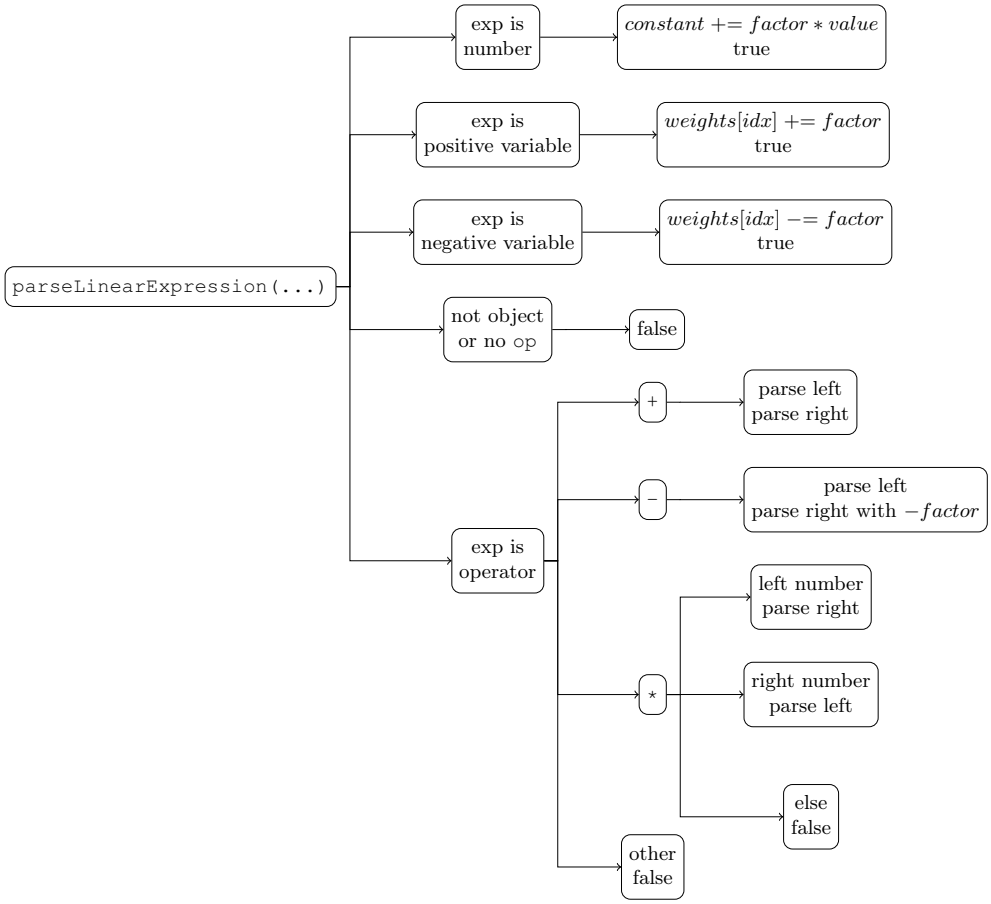

Figure 5.3: Decision tree for parsing linear expressions in JANI.

tree of the given expression. It requires the expression itself as input, as well as a mapping that stores the coefficients of the variables, a constant term, and a scaling factor. The scaling factor is used to propagate the coefficients correctly during recursive traversal. If the expression is a numeric constant, it is multiplied by the current factor and the result is added to the constant term. If the expression represents a variable, the corresponding coefficient in the weight map is increased or decreased depending on the sign of the occurrence. For compound expressions, the function distinguishes between different operators. In the case of addition, both operands are processed recursively with the same scaling factor, and their resulting coefficients and constant terms are added together. For subtraction, the left operand is processed with the current factor and the right operand with the negated factor, which effectively captures the difference between the two terms. Multiplication is only supported if one operand is a numeric constant. In this case, the constant is interpreted as a coefficient and combined with the current scaling factor. The function then recursively processes the remaining operand. If both operands are non-constant expressions, the function returns `false`, as such expressions are not linear. If the expression does not match any of the supported cases, the parsing fails. Overall, the function ensures that only

linear expressions composed of variables, numbers, additions, subtractions, and scalar multiplications are accepted and translated into a coefficient-based representation. An example for this function will be shown in Chapter 6.2.

5.1.4 Parsing different Linear Attributes

The parser has different functions for parsing the various attributes of hybrid automata. For each function which needed to be extended for parsing linear hybrid automata, the signature also needed to be adapted. Since the parser could only parse rectangular automata before, flows, guards, invariants, and properties were only represented as `VariableIntervalMaps`. To allow for the display of intervals and linear terms for these attributes, we replaced the `VariableIntervalMaps` with custom-defined maps. While these general maps differ in name, their definition is identical. This meant that saving as an interval or linear expression was possible in all functions.
In `parseFlow(...)`, a flow defined by the operator "*der*" can be expressed in two ways. It can be defined by a constant only, which is permitted for both rectangular and linear hybrid automata. This case was already supported in the existing implementation and therefore did not require any big modifications. However, complications would arise from the fact that these flows can occur in both types of automata. They can be specified as either intervals or linear expressions. Internally, RealySt requires these representations to be strictly separated, meaning that flows must be stored uniformly as either interval-based data types or linear terms, but not both simultaneously. This necessitates a design decision during parsing. Therefore, the representation is now determined based on whether the `--LTIAnalysis` option is enabled or not. If it is, flows are stored as linear flows. Otherwise, they are represented as intervals. Furthermore, the expression is no longer restricted to constants. In the previous implementation, only constant expressions were expected, as these were sufficient for rectangular automata. With the extension to linear hybrid automata, this assumption no longer holds, requiring an additional case to be handled. Consequently, we now check whether the expression is not a constant but linear and, if so, represent it as a linear flow using tuples of coefficients and variables.
The function `parseGuards(...)` is responsible for translating the guard expressions of transitions into an internal representation. Originally, this mapping was used for rectangular guards only, but now it is also used for linear guards. We have now extended the function so that it distinguishes between rectangular and linear guards. If the expression is of the form `linear_expression` $\leq$ `constant`, then the linear expression is analyzed and the result normalized to $\sum_i a_i * x_i \leq b - c$, so that the constant part is moved to the right-hand side. This matches the internal representation used by RealySt, where linear guards are stored as weighted variables together with a separate right-hand side value. The coefficients are stored in a `LinearFlow`. Additionally, the mirrored case, where the linear term is on the right-hand side, is handled similarly. Linear guards including equality constraints are not represented directly in a point interval like normal equality constraints since HyPro cannot handle equality constraints. Instead, they are translated into two inequalities, corresponding to the conjunction of $\leq$ and $\geq$, which are internally treated as separate constraints. If the expression does not match the linear pattern, the parser falls back to the original behavior and interprets the constraint as a rectangular guard, provided it consists of a single variable and a constant. In such cases, the constraint is stored as an interval.

The function `parseInvariant(...)` follows a similar structure but is applied to the time-progress conditions of locations while excluding any expressions involving derivatives. Apart from that, the extensions we made are identical to those in `parseGuards(...)`. The function differs depending on whether the invariant is rectangular or linear, the procedure for linear cases is the same.
The function `parseProperty(...)` is used to parse the goal conditions specified in the model. These properties are usually expressed as constraints on the state space and may appear within temporal logic formulas. As in the previous functions, conjunctions are handled recursively. Linear inequalities are handled in the same way as in the other functions. The difference here is that equalities have not been added, as they do not exist for rectangular cases.
Overall, the extensions made in `parseGuards(...)`, `parseInvariant(...)`, and `parseProperty(...)` generalize the parser from purely rectangular constraints to linear constraints. The key idea is to detect linear expressions, break them down into their coefficients and constants, and then normalise them into a uniform inequality form. Meanwhile, the original interval-based approach is retained for simpler expressions to ensure backwards compatibility. This approach allows linear hybrid automata to be represented within the existing parsing framework without altering its fundamental structure. The same design principle is consistently applied to flows via `parseFlow(...)`. By introducing a unified representation that supports both interval-based and linear dynamics, the parser is extended in a modular way. As a result, linear time-invariant behavior can be handled analogously to linear constraints, completing the transition from purely rectangular models to a more expressive class of hybrid automata.

5.1.5 Creating linear terms

To support the export of linear hybrid automata, the creation of linear terms had to be added as a separate building block in `RealySt2Jani.tpp`. For this purpose, the helper function `buildLinearExpressionTree(...)` was introduced. Its task is to transform linear expressions into a nested JANI expression tree. If the coefficient is zero, nothing happens and if it is one, only the variable is created. For every other pair, the function creates two nodes, one containing the coefficient and the other containing the variable, and connects them with the multiplication operator. If a constant is part of the term, it is also added separately. The resulting terms are then combined into a nested tree of addition operators. This process systematically translates internal linear or affine expressions into the binary JSON expression format defined by the JANI specification, thereby ensuring compatibility with standard JANI model representations.
This function was required in several parts of the export. In the function `locationsToJani(...)`, we needed to extend the creation of flows by linear and affine flows. Here, each row of the flow matrix corresponds to the derivative of one variable. The nonzero coefficients of the row are collected together with the corresponding source variables, while the constant part is taken from the affine offset. These components are then passed to `buildLinearExpressionTree(...)`, which constructs the right-hand side of the differential equation in JANI form.
We applied the same principle for affine resets in `transitionsToJani(...)`. A reset of the form $x' = R \cdot x + r$ is interpreted row by row in the same way, where $R \in \mathbb{R}^{n \times n}$ is a matrix describing the linear transformation of the variables, and $r \in \mathbb{R}^n$

is a vector representing constant terms. Each row defines the new value of one target variable as a linear combination of the old variables plus an optional constant. Again, the coefficients and constant are collected first and then translated into a JANI expression tree by `buildLinearExpressionTree(...)`. This makes it possible to export not only constant resets, but also general affine update functions.
For the construction of linear terms in guards, invariants and specifications, we needed to extend `createConstraintSetJani(...)` where attributes including constraints are generated in general. Here, we have included the linear case whereby, during export, each constraint row is converted into a list of weighted variables and a right-hand side bound. The left-hand side is then reconstructed as a linear expression tree, while the comparison with the bound is represented as a JANI inequality. This ensures that both rectangular and genuinely linear constraints can be represented uniformly.
Overall, the introduction of `buildLinearExpressionTree(...)` and the corresponding coefficient-based preprocessing made it possible to export affine flows, affine resets, and general linear constraints in a consistent way. This completed the extension from a purely rectangular export to one that supports the central elements of linear hybrid automata.

5.2 Extending Support for Specifications and Resets

During the extension of the parser for linear hybrid automata, we found other functionalities which were also missing and added them as well. We describe them in the following section.

5.2.1 Specification

The parser could already read multiple specifications. In this work, we mainly tested and extended this functionality. We extended the method `parseProperty AndCreateSpecification(...)` so that it can also read goal locations from the JANI-file. If a location is specified in the `states` block, its index is obtained and passed to `addSpecification(...)`. This allows specifications to refer to specific locations. Additionally, support for time bounds was added. If a property contains an upper time bound, it is extracted and stored in the specification using `setSpecificationTimebound(...)`. The specification struct needed to be extended by a new attribute to store the value. The reading worked without this extension but the export could not include upper time bounds. On the export side, the method `specificationsToJani(...)` was implemented, as this functionality was previously missing. It iterates over all specifications and converts them back into JANI format. All constraints of a specification are combined using conjunctions. Goal locations and time bounds are also written back into the new JANI-file.

5.2.2 Resets

Support for resets was not available in the parser before and was added in this work. During parsing, assignments in transitions are processed in the method `parseAndCreateTransition(...)`. Each assignment consists of a target variable and an expression describing its new value. If the expression is a number, it is interpreted as a constant reset. If it is a linear expression, it is parsed using

`parseLinearExpression(...)`, which extracts coefficients for variables and a constant term. The reset is then stored internally as an linear or affine function of the form $x' = \sum_i a_i x_i + c$ where the coefficients a_i represent the influence of the current variables x_i on the updated value, and c is a constant offset. This allows updates where the new value of a variable depends linearly on other variables. Assignments that contain probability distributions are ignored in this step, since stochastic behavior is handled separately.
For exporting, the method `transitionsToJani(...)` reconstructs these affine resets. The internal representation uses a matrix and a vector. Each row describes how one variable is updated. For each row, all non-zero coefficients are collected and converted into a JANI expression using `buildLinearExpressionTree(...)`. If a constant term exists, it is added to the expression. Resets that do not change a variable are ignored. All other resets are written as assignments in the JANI-file. With this extension, affine resets can now be parsed from JANI models and exported back into JANI format.

Chapter 6

Results

This chapter presents the results of the extensions we implemented in the JANI parser in REALYST. We tested our extensions with Modest to evaluate their compatibility with other tools.

6.1 Extending and testing the parser

As part of the bachelor's thesis, the JANI parser in REALYST was extended to support additional constructs of the JANI format. In particular, linear expressions can now be parsed and handled in a uniform way. This includes linear guards, invariants, flows, and specifications. Expressions are recursively processed and transformed into an internal affine representation consisting of variable coefficients and a constant term. Another important extension is the support for affine resets. Previously, resets were not handled by the parser. With the new implementation, resets in transitions can now be parsed as either constant or linear expressions. These are stored internally as affine functions and can also be exported back into JANI format. The handling of specifications was also improved. Goal locations and time bounds can now be parsed and stored, and specifications can be exported. Overall, linear hybrid automata can be read by the parser now. For example, the heating system from the beginning can be read and written as a JANI-file, the file is listed in the Appendix A. While the parser was extended by writing specifications during the creation of JANI-files, we also tested what happens if multiple specifications are defined during the analysis in REALYST. We found out by interpreting the probability during testing that the specifications are treated as disjunctions. The result is a union of the intervals the stochastic variable can have. We could see that if different traces were possible, the probability increased, otherwise it stayed the same.

6.2 Plotting with Modest

During the extension of the parser, maintaining compatibility with other tools supporting the JANI format was an important objective. Therefore, we evaluated whether our extensions remain compatible with existing toolchains. In particular, we chose the Modest Toolset [mod], as it has previously been shown to be compatible with our

implementation prior to the introduced extensions [Zum24].

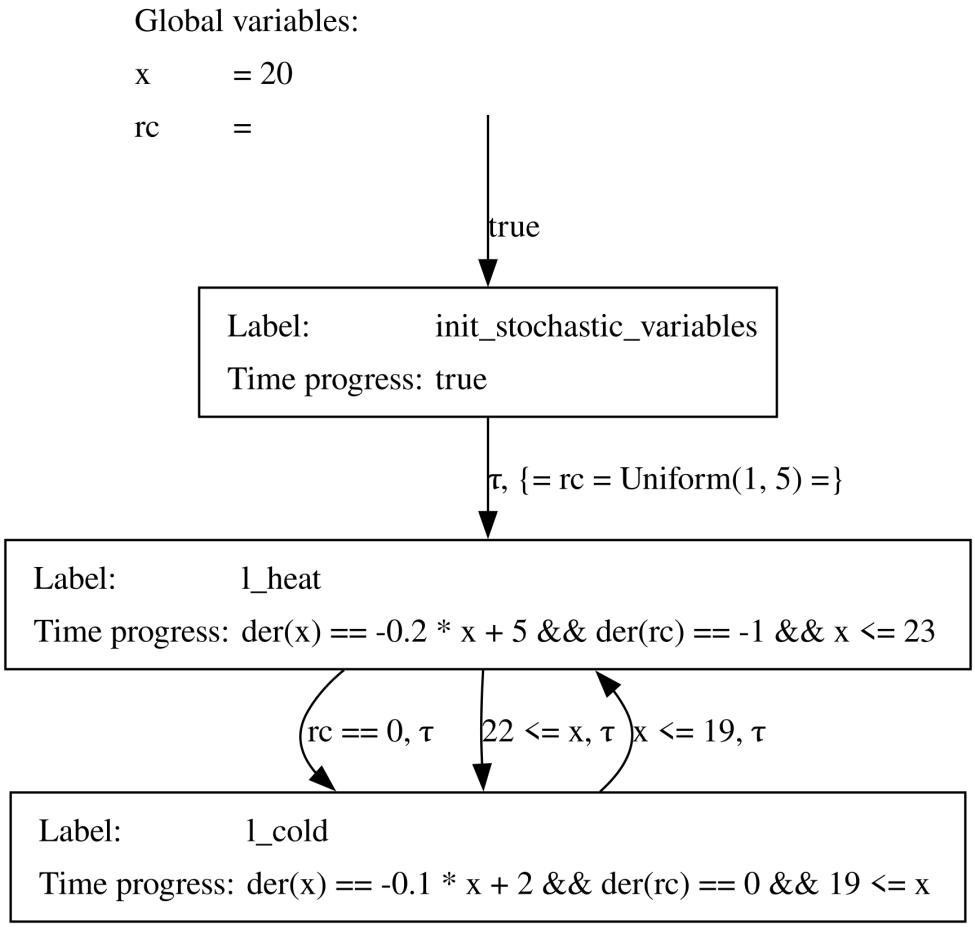

Figure 6.1: Example Heating System Plot created with Modest [mod]

We attempted to create a graphical representation of our heating example with random clocks, introduced in Chapter 2 in Figure 2.3, using Modest. Although the same automaton is represented, some differences in the plot in Figure 6.1 can be observed. The initialization of global continuous variables remains unchanged, whereas the handling of random clocks differs. This is due to the fact that stochastic variables are defined as continuous variables in JANI. As a consequence, an additional location is required to initialize these variables according to their distribution. This additional location is also included in the corresponding JANI-file, as shown in Appendix A. Apart from this difference in representation, the resulting automaton is semantically equivalent to the original one.
When we converted our JANI-file into a graphic with Modest, we observed that our extension for linear expressions is compatible with Modest, as the flows are correctly represented as linear functions in the graphical output. This makes Modest well suited

for visualizing linear hybrid automata. However, we also found that Modest does not support the declaration of goal locations in our approach. Since we could not find further information regarding how goal locations should be defined in JANI, we do not know if our approach will be suitable in the future. For now, it only works for the parser in REALYST.
In addition to this example, we created several JANI-files to test the correctness of the parser. For this, we parsed the test files in REALYST and then generated new JANI-files from the resulting models. Afterwards, we compared the input and output files to check whether they describe the same automata. This allowed us to verify that the parsing and export steps preserve the structure of the model. We also tested different linear expressions, resets, and specifications to ensure that the new features work as expected.

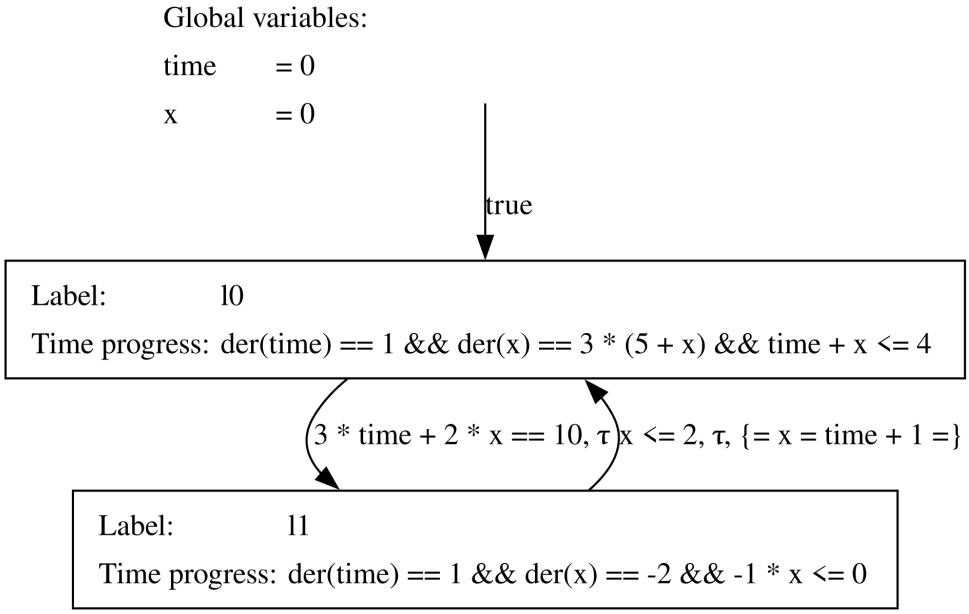

Figure 6.2: Example Linear Hybrid Automata Plot created with Modest [mod]

The automaton from one of our test JANI-files is represented in Figure 6.2. It contains a linear reset and guard, as well as a linear flow with nested expressions. To illustrate the parsing process, we show the parsing of the flow $der(x) = 3 * (5 + x)$ in location l_0 from the automaton shown in Figure 6.2 as an example. This flow is processed by the function `parseFlow(...)`, which detects that the variable is given on the left-hand side and the flow expression on the right-hand side. To parse the flow, it calls `parseLinearExpression(...)` with the term, an empty map `weights`, and the constant 0. The scaling factor is set to 1 by default. To process the linear expression, `parseLinearExpression(...)` first detects the outermost `"*"` operator and checks whether one of its operands is a constant. Since 3 is a number, the function recursively calls itself on the subexpression $(5 + x)$. At this point, `weights` is still empty, the constant remains 0, and the factor is updated to 3. The subexpression is then split at the `"+"` operator into 5 and x, which are both

processed with the same scaling factor. The constant 5 is multiplied by the factor 3, and the result 15 is added to the constant part. If additional constant terms were present, they would be accumulated in the same way. For the variable x, the value at its corresponding index in `weights` is increased by the factor 3. As a result, the expression is transformed into the linear form $3x + 15$, which can be stored in the internal representation `LinearFlow` used by RealySt.

In the output file, instead of $der(x) = 3 * (5 + x)$, the flow is defined as $3 * x + 15$, which is logically equivalent but written in a different form. This happens because the parser does not store the original syntactic structure of the expression. Instead, `parseLinearExpression(...)` transforms the expression into the internal linear representation used by RealySt, where coefficients and the constant part are stored separately. When the model is exported back to Jani, the function `buildLinearExpressionTree(...)` is used to reconstruct a Jani expression from this internal representation. It iterates over the stored variable coefficients and creates multiplication terms, for this flow the only multiplication is $3 * x$. These terms are then combined using addition operators. If a constant part is present, it is added to the expression tree as an additional term. Therefore, the internal representation of $3 * (5 + x)$ is exported as $3 * x + 15$. Although the expression is written differently in the output file, it describes the same linear flow. The difference only results from the normalization step during parsing and the reconstruction of the expression tree during export.

Chapter 7

Conclusion

In this bachelor thesis, the parser between the `jani-model` specification and RealySt was extended with additional functionality, focusing on linear expressions, affine resets, and improved handling of specifications.

7.1 Summary

The main contribution of this work is the extension of the parser to support linear hybrid automata. Previously, the parser was limited to rectangular automata. Therefore we extended it to handle general linear expressions represented as expression trees in Jani. This includes expressions such as sums, differences, and products with constants, which are recursively parsed and transformed into an internal linear representation. Another important addition is the support of affine resets. While resets were not handled before, the parser can now process assignments of the form $x := Ax + b$. These resets are parsed from Jani expressions into weighted variables and constants, and are then passed to the builder as affine transformations. Furthermore, the handling of specifications was improved. Although basic parsing was already possible, this work extends it by supporting goal locations and time-bounded properties. Specifications can now be translated into internal constraints and later reconstructed into Jani format. This allows a more complete round-trip between RealySt and Jani. In addition, the export functionality was extended accordingly. Linear flows, invariants, guards, and resets are now translated back into Jani expression trees. Additionally, specifications are now exported, this feature was missing completely before. This ensures that models using linear dynamics can be correctly represented in both directions. Overall, the parser now supports a significantly larger subset of the `jani-model` specification, especially with respect to linear hybrid automata. This extension is relevant in practice, as it allows RealySt to be used with a wider range of models that can be exchanged via Jani. Models with linear dynamics, in particular, can now be transferred between tools without manual adaptation, thus facilitating interoperability and reducing modelling effort. This enables practitioners to reuse existing models and apply different analysis tools more easily, thereby improving the efficiency and reproducibility of stochastic hybrid system analysis.

7.2 Outlook

Despite these improvements, several limitations remain. The parser currently supports only a subset of JANI expressions, and more complex or non-linear expressions are not handled. Extending the expression handling to cover a wider range of operators would improve robustness. Another limitation is the restricted support for multiple automata and more complex system structures. Currently, only a single automaton is processed. Future work could extend the parser to handle full networks of automata as supported by the JANI specification. In addition, compatibility with external tools such as the Modest Toolset could be further improved. This includes ensuring that all generated expressions are accepted and correctly interpreted by these tools. Finally, further testing and validation with more complex case studies would help to evaluate the correctness and performance of the parser. This would also support its use in comparing different modeling and analysis approaches for stochastic hybrid automata.

Bibliography

[Ábr21] Erika Ábrahám. Modeling and analysis of hybrid systems. *Lecture Notes*, 2021.

[AG 25] AG SKS. realyst-dev. `https://zivgitlab.uni-muenster.de/ag-sks/tools/realyst-dev`, 2025. GitLab Repository, Universität Münster.

[BDG+11] Thomas Brihaye, Laurent Doyen, Gilles Geeraerts, Joël Ouaknine, Jean-François Raskin, and James Worrell. On reachability for hybrid automata over bounded time. In *International Colloquium on Automata, Languages, and Programming*, pages 416–427. Springer, 2011.

[BDH+17] Carlos E Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. Jani: quantitative model and tool interaction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–168. Springer, 2017.

[BHR24] Pauline Blohm, Paula Herber, and Anne Remke. Towards quantitative analysis of simulink models using stochastic hybrid automata. In *International Conference on Integrated Formal Methods*, pages 172–193. Springer, 2024.

[DSÁR23] Joanna Delicaris, Stefan Schupp, Erika Ábrahám, and Anne Remke. Maximizing reachability probabilities in rectangular automata with random clocks. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 164–182. Springer, 2023.

[DSSR23] Joanna Delicaris, Jonas Stübbe, Stefan Schupp, and Anne Remke. Realyst: A c++ tool for optimizing reachability probabilities in stochastic hybrid systems. In *EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 170–182. Springer, 2023.

[HHHK13] Ernst Moritz Hahn, Arnd Hartmanns, Holger Hermanns, and Joost-Pieter Katoen. A compositional modelling and analysis framework for stochastic hybrid systems. *Formal Methods in System Design*, 43(2):191–232, 2013.

[HSRÁ17] Jannik Hüls, Stefan Schupp, Anne Remke, and Erika Ábrahám. Analyzing hybrid petri nets with multiple stochastic firings using hypro. In *Proceedings of the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 178–185, 2017.

[hyp25] realyst-dev. `https://github.com/hypro/hypro`, 2025. GitLab Repository, RWTH Aachen.

[LG09] Colas Le Guernic. *Reachability analysis of hybrid systems with linear continuous dynamics*. PhD thesis, Université Joseph-Fourier-Grenoble I, 2009.

[mod] Modest toolset. `https://www.modestchecker.net/`. Accessed: 2026-04-16.

[RHH25] Anne Remke, Boudewijn R Haverkort, and Johann L Hurink. Running. christel: A stochastic hybrid case-study optimizing battery pack. *Principles of Formal Quantitative Analysis: Essays Dedicated to Christel Baier on the Occasion of Her 60th Birthday*, page 382, 2025.

[SÁMK17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. Hypro: A c++ library of state set representations for hybrid systems reachability analysis. In *NASA Formal Methods Symposium*, pages 288–294. Springer, 2017.

[Sch19] Stefan Schupp. *State set representations and their usage in the reachability analysis of hybrid systems*. PhD thesis, Dissertation, RWTH Aachen University, 2019, 2019.

[WRÁ24] Lisa Willemsen, Anne Remke, and Erika Ábrahám. (de-)composed and more: Eager and lazy specifications (camels) for stochastic hybrid systems. In *Principles of Verification: Cycling the Probabilistic Landscape: Essays Dedicated to Joost-Pieter Katoen on the Occasion of His 60th Birthday, Part III*, pages 309–337. Springer, 2024.

[Zum24] Stefan Zumdick. Implementierung eines Parsers für die JANI-Spezifikation in RealySt, 2024.

[ZWZ+19] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794. IEEE, 2019.

Appendix A

Heating Example

```
{
  "$schema": "../../jani-schema.json",
  "jani-version": 1,
  "name": "heating_rc_example",
  "type": "sha",

  "variables": [
    {
      "name": "x",
      "type": "continuous",
      "initial-value": 20.0
    },
    {
      "name": "rc",
      "type": "continuous"
    }
  ],

  "automata": [
    {
      "name": "heating_automaton",

      "initial-locations": [
        "init_stochastic_variables"
      ],

      "locations": [
        {
          "name": "l_heat",
          "time-progress": {
            "exp": {
              "left": {
                "left": {
                  "left": { "op": "der", "var": "x" },
```

```
35                    "op": "=",
36                    "right": {
37                      "left": {
38                        "left": -0.2,
39                        "op": "*",
40                        "right": "x"
41                      },
42                      "op": "+",
43                      "right": 5.0
44                    }
45                  },
46                  "op": "∧",
47                  "right": {
48                    "left": { "op": "der", "var": "rc" },
49                    "op": "=",
50                    "right": -1.0
51                  }
52                },
53                "op": "∧",
54                "right": {
55                  "left": "x",
56                  "op": "≤",
57                  "right": 23.0
58                }
59              }
60            }
61          },
62          {
63            "name": "l_cold",
64            "time-progress": {
65              "exp": {
66                "left": {
67                  "left": {
68                    "left": { "op": "der", "var": "x" },
69                    "op": "=",
70                    "right": {
71                      "left": {
72                        "left": -0.1,
73                        "op": "*",
74                        "right": "x"
75                      },
76                      "op": "+",
77                      "right": 2.0
78                    }
79                  },
80                  "op": "∧",
81                  "right": {
82                    "left": { "op": "der", "var": "rc" },
83                    "op": "=",
```

```
84                    "right": 0.0
85                  }
86                },
87                "op": "∧",
88                "right": {
89                  "left": 19.0,
90                  "op": "≤",
91                  "right": "x"
92                }
93              }
94            }
95          },
96
97          {
98            "name": "init_stochastic_variables",
99            "comment": "Necessary to initialize stochastic
    variables"
100         }
101       ],
102
103       "edges": [
104         {
105           "location": "l_heat",
106           "guard": {
107             "exp": {
108               "left": "rc",
109               "op": "=",
110               "right": 0.0
111             }
112           },
113           "destinations": [
114             {
115               "location": "l_cold"
116             }
117           ]
118         },
119         {
120           "location": "l_heat",
121           "guard": {
122             "exp": {
123               "left": 22.0,
124               "op": "≤",
125               "right": "x"
126             }
127           },
128           "destinations": [
129             {
130               "location": "l_cold"
131             }
```

```
132             ]
133           },
134           {
135             "location": "l_cold",
136             "guard": {
137               "exp": {
138                 "left": "x",
139                 "op": "≤",
140                 "right": 19.0
141               }
142             },
143             "destinations": [
144               {
145                 "location": "l_heat"
146               }
147             ]
148           },
149           {
150             "location": "init_stochastic_variables",
151             "destinations": [
152               {
153                 "assignments": [
154                   {
155                     "ref": "rc",
156                     "value": {
157                       "args": [
158                         1.0,
159                         5.0
160                       ],
161                       "distribution": "Uniform"
162                     }
163                   }
164                 ],
165                 "location": "l_heat"
166               }
167             ]
168           }
169         ]
170       }
171     ],
172
173     "system": {
174       "elements": [
175         {
176           "automaton": "heating_automaton"
177         }
178       ]
179     }
180   }
```