Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

MASTER OF SCIENCE THESIS

SOLVING INITIAL-STATE NON-DETERMINISM IN STOCHASTIC LINEAR HYBRID AUTOMATA WITH UNDERAPPROXIMATIVE BACKWARDS REFINEMENT

Yvonne Heimowski

Communicated by
Prof. Dr. Erika Ábrahám

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

Additional Advisor:
József Kovács

Aachen, April 17, 2026

Abstract

In recent decades, an increasing number of industrial and real-life safety-critical processes have been automated and are now predominantly controlled by programs, increasing the need for formal safety and correctness verification techniques. Many such processes can be categorized as Hybrid Systems which combine continuous and discrete components. Hybrid Automata provide a modeling formalism for such Hybrid Systems, which enable various safety checks and analysis on the systems. Many Hybrid Systems, and consequently the associated Hybrid Automata allow various sources of non-determinism in their formalism. Various ways have been proposed to deal with the issue of non-determinism during analysis. One source of non-determinism in Hybrid Automata is posed by an initial state that is not fixed to a unique value but chosen non-deterministically from a set of possible values. The choice of the initial set may influence the correctness of the execution of a Hybrid System. To tackle this issue, we propose a method to probabilistically solve the non-determinism introduced by the initial set by utilizing backwards refinement. The approach extends existing methods for flowpipe construction-based forward reachability analysis. The results of the forward analysis are used for a backwards refinement, reconstructing the subset of the initial set that led to a goal state. The feasibility of our approach is illustrated by presenting the results of the implementation of our method on various benchmarks.

Contents

Acronyms

HA Hybrid Automata. iii, 7, 9–15, 20, 57, 59

HS Hybrid System. 7, 8, 10, 12, 15, 33

LHA Linear Hybrid Automaton. 7, 13–15, 19–23, 33, 34, 56, 60, 63

ODE Ordinary Differential Equation. 7, 10, 13, 14, 20, 27, 28

RA Rectangular Automata. 13, 15, 16, 18, 20, 59

RAC Rectangular Automata with Random Clocks. 18, 19, 59–61

RAE Rectangular Automata with Random Events. 15–18, 59

SLHA Stochastic Linear Hybrid Automaton. 19, 23, 25, 31, 45, 46, 48, 49, 52, 54, 57, 60, 61, 64

Chapter 1

Introduction

Over the course of recent decades, automation has increasingly been implemented in industrial and real-life processes, such that many of them are now controlled by programs. As we entered a time of automation, ensuring the safety and correctness of such processes has been gaining importance, whether the system is an Industry 4.0 robot that must remain within its defined range of motion or an autopilot system controlling the altitude of an aircraft. Without safety assurances, such systems cannot be trusted in productive environments. Although testing the final system is an important step in the regulation of safety-critical systems, we cannot rely on this method to test the entire state space or behavior of the system. To verify safety of the complete state space in an earlier development phase, formal verification techniques can be employed [Sch19].
Many of the aforementioned safety-critical systems fall into the category of Hybrid System (HS). A HS consists of states with continuous dynamics that are connected through discrete state changes. A very simple example of such a system would be a thermostat [ACH+95]. The system is given a minimum and maximum temperature and measures the current room temperature. When the room temperature is below the maximum temperature, the thermostat is in the *heating* state. While in the *heating* state, the temperature continuously increases. Once the maximum temperature is reached, the state switches to the *idle* state, where the temperature continuously decreases. It stays in this state until the minimum temperature is reached, at which point it switches back to *heating*.

HS can be modeled as Hybrid Automata (HA), enabling the application of correctness and safety verification techniques developed for HA. With HA, we can compute e.g. if certain states in the system are reachable. Proving correctness in the case of the thermostat would be to show that the temperature never reaches above the maximum temperature or below the minimum. Depending on the dynamics in the continuous states, the HA can be analyzed with different algorithms. The dynamics of the values in the system may be defined by e.g. constants, intervals, or linear expressions, allowing for different ranges of expressivity. In Linear Hybrid Automaton (LHA), the dynamics of the system are defined by systems of linear Ordinary Differential Equation (ODE) allowing for a high degree of expressivity [Sch19].

Real-world HS are often exposed to uncertainty in their environment leading to nondeterministic behavior. Such uncertainty can show up as unpredictably occurring

events. The aforementioned thermostat controls the temperature of a room. However, a window in the room may be opened, thus changing the dynamics which describe the change of the temperature over time. The outside temperature is out of our control, making it entirely possible that the temperature falls below the minimum. We must ensure that the system responds correctly to these events and recovers to a valid temperature. HS with randomly occurring state changes are referred to as Stochastic Hybrid Systems. Safety and correctness checks for such systems need to account for non-determinism introduced by the uncertainty. One way of dealing with non-determinism is by utilizing reachability probabilities, which state the likelihood that the system will execute successfully [DSÁR23].
Another source of non-determinism in HS is introduced by a flexible initial state. In some systems, the initial state of the system may be unique. For example, an Industry 4.0 robot may be designed to perform a specific task, such as grabbing and placing objects. The objects to be grabbed may always be of the same size and placed in the same spot. In this scenario, the initial state of the work process of the robot is uniquely defined. However, the object may also be of varying sizes and placed somewhere in a defined area. In such a scenario, there are many possibilities for the initial state of the work process. In order to model and formally verify such a system, we need to ensure that the entire range of possible initial states is included, since some combinations of size and placement of the objects may lead to an erroneous state in the system. This type of non-determinism is referred to as initial-state non-determinism. In this thesis, we explore an approach to solve this type of non-determinism probabilistically in such systems. We propose a method to compute the probability that a set of goal states can be reached from the initial set. Our approach builds on existing methods for forward analysis and extends them by adding backwards refinement to reconstruct the subset of the initial set that can lead to a goal state.

We start by providing the theoretical foundations needed to understand the contents of this thesis in Chapter 2. Then, in Chapter 3, we explain the methodology of our approach. In Chapter 4, we introduce the tools used for the realization of our approach before diving into the details of the implementation in Chapter 5. In Chapter 6, the results of our proposed method are shown on various benchmarks. Finally, we finish with a discussion of related and future work in Chapter 7 and present our conclusion in Chapter 8.

Chapter 2

Preliminaries

In this chapter, we present the theoretical background and formalisms necessary for the thesis. First, we introduce some mathematical terms before delving into different forms of HA. We continue by introducing the reachability problem and flowpipe construction as an approach to solve it. Finally, we describe various forms of nondeterminism in HA.

2.1 Mathematical Preliminaries

In this section, we introduce some mathematical terms that are used throughout the thesis.

Linear Term and Inequality

A linear term describes an expression of the form

$$k_0 + k_1 x_1 + ... + k_n x_n$$

with x_0,..,x_n, k_0,...$k_m \in \mathbb{R}$. A linear inequality is an expression of the form

$$t_1 \leq t_2$$

with t_1, t_2 as linear terms [Hen00].

Affine Transformation

An affine transformation for the d-dimensional set S, is defined as

$$\{A \cdot s + b \mid s \in S\}$$

with $A \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$. An affine transformation expresses a composition of linear transformations, such as rotations, scalings and shearings defined by the matrix A, and a translation defined by the vector b [Sch19].

Ordinary Differential Equation

An ODE with the vector $x = (x_0,..x_{d-1})^T$ is defined as

$$\dot{x} = \frac{dt}{dx} f(x) \tag{2.1}$$

where $\dot{x}$ denotes the derivative of x over time t with $f : \mathbb{R}^n \to \mathbb{R}^n$.
We can represent Equation 2.1 as

$$\dot{x} = Ax$$

with $A \in \mathbb{R}^{d \times d}$ and use the matrix exponential as

$$x(t) = e^{At} x(0)$$

to compute x after one timestep t [Egg14].

HPolytope

An HPolytope describes a convex polytope in *H-representation*. A d-dimensional Hpolytope is represented by a matrix $A \in \mathbb{R}^{m \times d}$ and a vector $b \in \mathbb{R}^m$

$$H = \{x \in \mathbb{R} \mid Ax \leq b\}$$

defining a convex set

$$P = \bigcap_{i=0}^{m-1} h_i$$

as the intersection of a finite number of half-spaces $\{h_0,..h_{n-1}\}$. Each halfspace is defined as

$$h_i = \{x \in \mathbb{R}^d \mid a_{i,\_} \cdot x \leq b_i\}$$

with $a_i,\_$ as the i-th row of A and b_i as the i-th value of b [Sch19].

2.2 Hybrid Automata

A HS combines the behavior of discrete and dynamic systems. Over time, the system evolves according to the definition of the continuous dynamic components until a discrete transition is triggered, causing an instantaneous state change. A HS can be a physical process, such as bouncing ball, that changes its dynamic, and therefore its state, whenever it bounces off the ground, or an industrial application or digital controller in a dynamic environment [ACH+95, Sch19].

HA define a modeling formalism of HSs to enable further analysis of the system using formal methods. The states of the HSs can be encoded as a set of *locations* with the transitions between them modeled as a set of *jumps*. A HS generally observes a set of values which define which state of the system is in and if the state should be changed. The observed values are modeled by set of *variables* in the HA. In each state of the HS, the observed values may behave differently. Looking at the example of the thermometer introduce in Chapter 1, the temperature changes differently whether the system is in the *heating* or *idle* state . The rate of change of the variables in the HA is defined by a function *flow* on the variables for each location. The values in the system determine whether a state is valid or should be changed. These conditions are

encoded as a set of *invariants* on the locations. They define the valid valuations of the variables to remain in a location. The values may also limit whether a transition into a different state is possible. This is modeled by a set of *guards* on the jumps. They restrict the valuations of the variables allowed for a certain jump to be taken. When a transition to another state in the system is performed, the values may be impacted. Hybrid automata account for these value changes through *resets* defined on the jumps. A reset defines a function over the variables, which determines how the value of the variables changes when the jump is taken. Each jump defines its source and target location, its guard, and a reset [Sch19].

To formally define HA, we denote the set of all predicates $Pred_X$ over set the real-value variables X as quantifier-free arithmetic formulas containing variables from X. Different subcategories of HA restrict the set of functions in the underlying arithmetic theory. We further define the subset of the predicated $Pred_{X_1,X_2}$ as *assignment predicates*, defining conjunctions of constraints of the form $x \sim e$ with $x \in X_2$, e is an expression of variables in X_1 and $\sim \in \{<, \leq, =, >, \geq\}$. The assignment predicate φ is specified as

$$\varphi = \bigwedge_{i=0}^{n-1} (x_i \sim_i c_i) \in Pred_{X_1,X_2}$$

For $x \in X_2$, we define

$$\varphi[x] = \bigwedge_{i:x_i=x} (x \sim_i c_i)$$

$\nu \models \varphi$ expresses the variable valuation $\nu \in \mathbb{R}^{|X|}$ for the variables X as a model of $\varphi \in Pred_X$. We define a *state* $\sigma \in \sum = (Loc \times \mathbb{R}^d)$ of a HA as a combination of a location l from Loc and a variable valuation $\nu \in \mathbb{R}^d$. We refer to a set of valuations V with the same location as *state sets* $(l,V) \in \{(l,\nu) \mid \nu \in V\}$ [Sch19].

Definition 2.2.1 (Syntax of a Hybrid Automata [Sch19])**.** A d-dimensional Hybrid Automata is a tuple *H = (Loc, Var, Sync, Flow, Inv, Jump, Init)* with

- A finite set *Loc* of locations
- A finite ordered set $Var = \{x_0,..x_{d-1}\}$ of dimension d of real-valued variables. $\dot{Var} = \{\dot{x}_0,..\dot{x}_{d-1}\}$ represents the first derivatives during continuous change and $Var' = \{x'_0,..x'_{d-1}\}$ represents the values after a discrete change.
- A finite set *Sync* of synchronization labels including the *stutter label* $\tau \in Sync$.
- Function $Flow : Loc \to Pred_{Var,\dot{Var}}$ specifies the flow for each location.
- Function $Inv : Loc \to Pred_{Var}$ defines an invariant to each location.
- $Jump \subseteq (Loc \times Sync \times Pred_{Var} \times Pred_{Var,Var'} \times Loc)$ is a finite set of jumps. A jump is defined as $(l,a,g,r,l') \in Jump$ with l as the source location, a as the synchronization label, g as the guard, r as its reset and l' as the jumps target location.
- Function $Init : Loc \to Pred_{Var}$ defines an initial predicate for each location.

We omit synchronization labels unless they are relevant for the specific context.
The behavior of a HA is defined through operational semantics.

Definition 2.2.2 (Operational Semantics of a Hybrid Automata)**.** The one-step semantics of a hybrid automaton $H = (Loc,Var,Sync,Flow,Inv,Jump,Init)$ is defined by the following rules [Sch19]:

$$\frac{\begin{array}{c} l \in Loc \;\; v,v' \in \mathbb{R}^d \;\; f : [0,\tau] \to \mathbb{R}^d \\ \partial f/\partial t = \dot{f} : (0,\tau) \to \mathbb{R}^d \;\; f(0) = \nu \; f(\tau) = \nu' \\ \forall \epsilon \in (0,\tau). f(\epsilon), \dot{f}(\epsilon) \models Flow(l) \\ \forall \epsilon \in [0,\tau], f(\epsilon) \models Inv(l) \end{array}}{(l,\nu) \xrightarrow{\tau} (l',\nu')} \qquad \texttt{Rule}_{\texttt{flow}}$$

$$\frac{\begin{array}{c} j = (l,g,r,l') \in Jump \;\; l,l' \in Loc \;\; \nu,\nu' \in \mathbb{R}^d \\ \nu \models g \;\; \nu,\nu' \models r \;\; \nu' \models Inv(l') \end{array}}{(l,\nu) \xrightarrow{j} (l',\nu')} \qquad \texttt{Rule}_{\texttt{jump}}$$

We define a *run* π of a HA as a ordered sequence of states which are connected by time and discrete steps according to the rules $\texttt{Rule}_{\texttt{flow}}$ and $\texttt{Rule}_{\texttt{jump}}$ in Definition 2.2.2.

In Chapter 1, we introduced the thermostat as a simple example for a HS. Figure 2.1 shows the HA modeling it. M and m denote the maximum and minimum temperatures, respectively. The automaton has two locations, *heating* and *idle* and uses one variable, x, representing the current temperature. Location *heating* defines a flow function $\dot{x}$ and an invariant. The flow function assigns a constant flow of 1 to x, and the invariant states that the automaton can only remain in the location while the value of x is at most the maximum temperature. The location *idle* also defines a flow function $\dot{x}$ and an invariant. Here, the flow function defines a constant flow of -1 for x , and the invariant states that x may not go below the minimum temperature. There are two transitions, one from *heating* to *idle*, and vice versa. Both define a guard restricting x to the maximum or minimum temperature, respectively. Neither transition defines a reset.

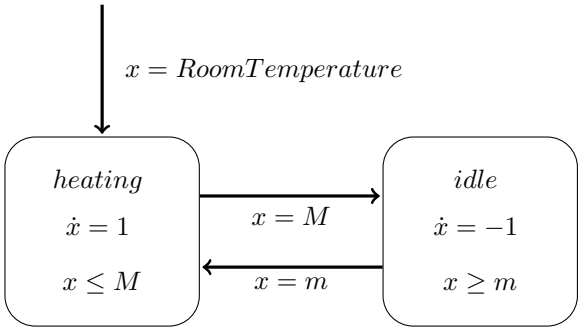


Figure 2.1: Simple Hybrid Automata modeling a thermostat [ACH+95].

2.2.1 Rectangular Automata

Rectangular Automata (RA) are a subset of HA. In RA the predicates for invariants, guards, flows and resets are encoded by rectangular sets. A rectangular is defined by conjugation of constraints which compare variables to constraints. We use the notation $x \in [a,b]$ to denote the constraints $a \leq x \ \wedge \ x \geq b$ [Kop96, Sch19].
We define $\mathbb{I}$ to be the set of all real-valued intervals of the form [a,b], [a,∞],(-∞,b] or (-∞,∞) for some a,b ∈ $\mathbb{Q}$

Definition 2.2.3 (Syntax of Rectangular Automata [DRÁ+25])**.** A Rectangular Automata is a tuple $H = (Loc, Var, Inv, Init, Flow, Jump)$ with *Loc, Var* as defined in Definition 2.2.1 and

- Function $Inv : Loc \to \mathbb{I}^d$ assigns an invariant to each location
- Function $Init : Loc \to \mathbb{I}^d$ assigns initial states with $Init(l) \subseteq Inv(l)\ \forall l \in Loc$ and $\exists l \in Loc : Init(l) \neq \emptyset$
- Function $Flow : Loc \to \mathbb{I}^d$ assigns flow rates to each location with $Flow(l) \neq \emptyset\ \forall l \in Loc$
- Finite set $Jump \subseteq Loc \times \mathbb{I}^d \times (\mathbb{I} \cup \{id\})^d \times Loc$ of jumps $j = (l, guard, reset, l') \in Jump$ with source location $l \in Loc$, *guard* defining the guard of the jump , *reset* defining the reset of the jump and and target location $l' \in Loc$ satisfying that $\forall x \in Var, guard_{|x} \neq \emptyset$ and if $reset_{|x} =$ id then $Inv(l)_{|x} \cap guard_{|x} \subseteq Inv(l')_{|x}$, else if $reset_{|x} \neq id$ then $reset_{|x} \cap Inv(l')_{|x} \neq \emptyset$

The defined conditions of *Init* and *Flow* ensure that the automaton is *non-blocking*, meaning that the possibility to execute a jump does not rely on the invariants of the locations but is solely dependent on the jump's guard.

Figure 2.2 shows a non-blocking RA. The automaton consists of three locations $\{l_0, l_1, l_2\}$ and two variables $\{a, b\}$. The flow of each variable in each location and the guards of all jumps are defined over rectangular sets.

2.2.2 Linear Hybrid Automata

In literature, the term LHA is typically used to describe one of two categories of HA. LHA with linear behavior [ACH+95] and LHA with non-linear behavior [Hen00, Sch19]. In this thesis, we will focus on LHA with nonlinear behavior, which is the more expressive class of the two and also more expressive than RA.

LHA with non-linear behavior: The automaton's flows, resets, guards and invariants are defined by linear expressions. The linear expressions are defined over the set of variables *Var* in the automaton. Guards and invariants can be expressed as linear inequalities of the kind $Ax \sim b$, resets are defined as affine mappings of the form $x' = Ax + b$ with $A \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$. The flows of the locations can be expressed as systems of linear ODEs

$$\dot{x} = Ax$$

with $x = (x_0, ..x_{d-1})^T$ as the variables of the given HA , $A \in \mathbb{R}^{d \times d}$. We can define the flow in the form $\dot{x} = Ax + b$ by adding a zero flow for the constant vector as an artificial dimension to transform it into the form presented above.

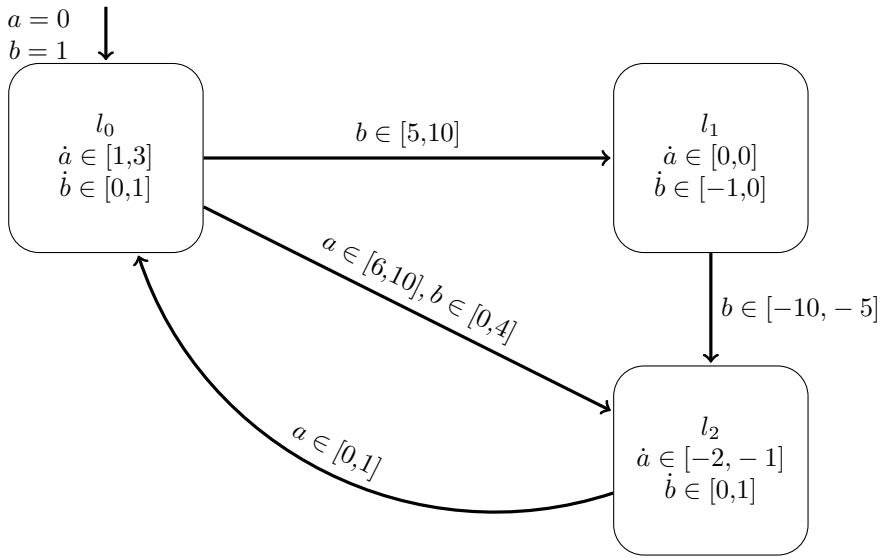


Figure 2.2: Rectangular Automaton with three locations and two variables.

We refine Definition 2.2.1 for LHA. We define ψ as a system of linear inequalities of the form $Ax \leq b$ with $A \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$ over *Var* and ξ as a system of linear ODEs over *Var* [Sch19].

Definition 2.2.4 (Linear Hybrid Automaton)**.** A Hybrid Automata is a tuple $H =$ *(Loc, Var, Flow, Inv, Jump, Init)* with

- Function $Inv : Loc \rightarrow \psi$ assigns a system of linear inequality as the invariant to each location.
- Function $Flow : Loc \rightarrow \psi$ assigns a system of linear ODE as the flow rate to each location.
- Finite set Jump $\subseteq$ Loc $\times \psi \times (\xi \cup id) \times$ *Loc*. A jump is defined as $(l,g,r,l') \in Jump$ with $l \in Loc$ as the source location, $l' \in Loc$ is the destination location, g as the guard as a system of linear constraints and r as the reset defining a system of linear ODEs.

Figure 2.3 shows a LHA with with nonlinear behavior. The tupel $H_l = (Loc_l, Var_l, Flow_l, Inv_l, Jump_l, Init_l)$ defines the automaton with

- Loc_l: $\{l_0,l_1\}$
- Var_l: $\{x,y\}$
- $Flow_l$: $\{l_0 \rightarrow \begin{bmatrix} 2 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 3 \\ 0 \end{bmatrix}, l_1 \rightarrow \begin{bmatrix} -0.5 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}\}$
- $Invariant_l$: $\{l_0 \rightarrow \begin{bmatrix} -1 & -1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \end{bmatrix}, l_1 \rightarrow \begin{bmatrix} 0 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 0 \\ 0 \end{bmatrix}\}$

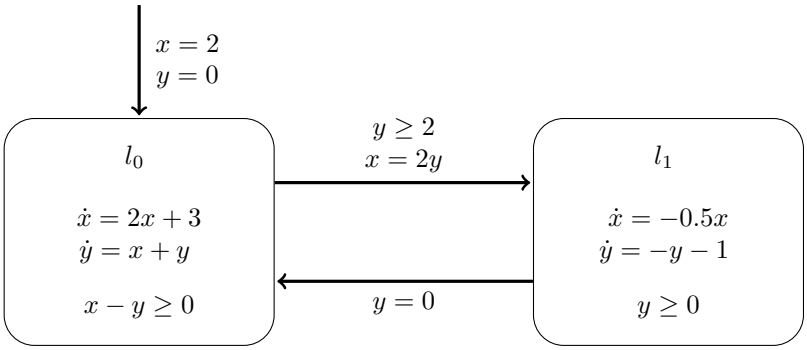


Figure 2.3: Linear Hybrid Automata with two locations $\{l_0,l_1\}$ and two variables $\{x,y\}$.

- $Jump_l$: $\{(l_0, \begin{bmatrix}0 & 0\\0 & -1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} \leq \begin{bmatrix}0\\-2\end{bmatrix}, \begin{bmatrix}0 & 2\\0 & 1\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix}, l_1),$
  $(l_1, 0, \begin{bmatrix}1 & 0\\0 & 0\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} + \begin{bmatrix}0\\0\end{bmatrix}, l_0)\}$
- $Init_l$: $l_0 \rightarrow \begin{Bmatrix}2\\0\end{Bmatrix}$

When we refer to LHA in this thesis, we refer to LHA with nonlinear behavior, unless otherwise specified.

2.3 Stochastic Hybrid Automata

Complex HSs often deal with uncertainty in their environment that causes state changes to occur unpredictably or non-deterministically. To model them, we can extend HA with stochasticity. In Stochastic Hybrid Automata we include these uncertainties as random events. Random events occur at random times during system execution. When a random event occurs, it triggers a stochastic jump in the automata [DK05, DRÁ+25, BBH06]. The aforementioned thermostat can be extended with stochasticity. In a room whose temperature is controlled by the thermostat, it may occur, that a windows is opened. When a windows is open, the dynamics of the the temperature change. The system needs to account for the imposed uncertainty since the temperature may go below the minimum temperature, violating previous correctness constraints.

2.3.1 Stochasticity in Rectangular Automata

Delicaris et al. defined an approach to model stochasticity in Stochastic Rectangular Automata by including random events syntactically, therefore optimizing reachability analysis [DSÁR23, DRÁ+25].

Rectangular Automata with Random Events

Rectangular Automata with Random Events (RAE) were defined by Delaris et al. as an extension of RA. RA only model nonstochastic jumps, whereas RAE also contains

stochastic jumps. While the time point of deterministic jumps is a matter of guard satisfaction, the time point of stochastic jumps is chosen randomly based on some predefined continuous distributions. For this purpose, we introduce *random events* that are each associated with a set of jumps [DRÁ+25].
$\mathbb{F}$ denotes a probability distribution over an interval $\mathbb{I}$.

Definition 2.3.1 (Syntax of Rectangular Automata with Random Events [DRÁ+25])**.** A RAE is a tuple $A = (H,Lab,Distr,Event)$ with H a rectangular automaton as defined in Definition 2.2.3

- a finite ordered set $Lab = \{r_0,..,r_{d_R-1}\}$ of random events
- a function $Distr : Lab \to \mathbb{F}$ from which an expiration time is sampled
- function Event : Jump $\to (Lab \cup \{\perp\}$ satisfying that two jumps $j = (l,guard,reset,l') \in Jump$ and $j' = (l,guard',reset',l'') \in Jump\backslash\{j\}$ with common source location and identical random event label Event(j) = Event(j') $\in$ Lab have disjoint guards. We call a jump $j \in Jump$ stochastic if and only if $Event(j) \in Lab$, and otherwise non-stochastic.

Remark. H must satisfy the fact that when we remove all stochastic jumps from *Jump*, the resulting RA is non-blocking.

Remark. Including identical jumps in different random events is not allowed for notational simplification.

As mentioned above, random events *Lab* of RAE gather jumps under a label. When a stochastic jump is triggered, the associated random event occurs. The time point when a stochastic jump is executed is determined by the function *Distr*. *Distr* assigns a continuous distribution for each $r \in Lab$ and from it samples an expiration time $s_{|r}$. Sampling happens once initially and after every occurrence of the random event r. We measure how long a jump has been enabled since the beginning of the run, and when a jump of random event r has been enabled for $s_{|r}$ time units, the event can occur, and the jump can be taken. After that, the expiration time of r $s_{|r}^{new}$ is resampled and the event can re-occur when a jump j with $Event(j) = r$ has been enabled for $s_{|r}^{new}$ after the last occurrence of r. To measure the time of enabledness semantically, we introduce logical stopwatches - referred to as random clocks - with value $\mu_{|r}$. We define a *state* σ in a RAE as the tuple $\sigma = (l,\nu,\mu,s) = Loc \times \mathbb{R}^d \times \mathbb{R}^d \times \mathbb{R}^d$ with the current location l, the valuations ν for the continuous variables, the duration of enabledness $\mu = (\mu_0,...\mu_{d_R-1})$ of the jumps and the expiration times $s = (s_0,...s_{d_R-1})$ of the random events [DRÁ+25].

The behavior of RAE is defined by operational schematics. Hereby, we denote that for the the time duration $t \in \mathbb{R}_{\geq 0}$ and $rate \in Flow(l)$, we define a time step $(l,\nu) \xrightarrow{t,rate}_H (l',\nu')$ with ν' uniquely determined from ν, t and $rate$. The operational schematics in Definition 2.3.2 specify the rules $\texttt{Rule}_{Jump_N}$ to define the execution of nonstochastic jumps, $\texttt{Rule}_{Jumps_S}$ to define the execution of stochastic jumps and $\texttt{Rule}_{Flow}$ to define the execution of flows with time.

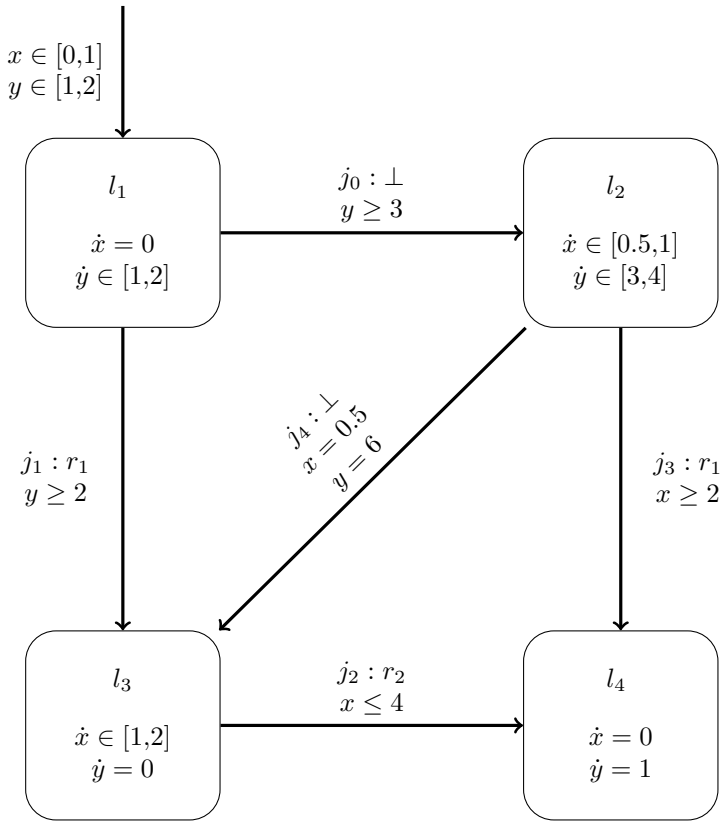


Figure 2.4: RAE with random events r_1,r_2, the stochastic jumps j_1, j_2, j_3 and deterministic jumps j_0, j_4.

Definition 2.3.2 (Operational semantics for RAE $\mathcal{E}$ [DRÁ+25])**.**

$$\frac{(l,\nu) \xrightarrow{j}_H (l',\nu')\ Event(j) = \bot}{(l,\nu,\mu,s) \xrightarrow{j}_\epsilon (l',\nu',\mu,s)} \quad \texttt{Rule}_{Jump_N}$$

$$\frac{(l,\nu) \xrightarrow{j}_H (l',\nu')\ Event(j) = r \in Lab\ \mu_{|r} = s_{|r}\ \mu'_{|r} = 0 \quad s'_{|r} \in supp(Distr(r))\ \forall r' \in Lab \setminus \{r\}.\mu'_{|r'}\ \wedge\ s'_{|r'} = s_{|r'}}{(l,\nu,\mu,s) \xrightarrow{j}_\epsilon (l',\nu',\mu',s')} \quad \texttt{Rule}_{Jump_S}$$

$$\frac{(l,\nu) \xrightarrow{t,rate}_H (l',\nu')\ \ \mu' = \mu + t \cdot Flow^{\mathcal{E}}_R(l,\nu,rate)\ \ \mu' \leq s \quad \forall 0 < t',t'' < t.InvariantConditions(l, \nu + t' \cdot rate, \nu + t'' \cdot rate)}{(l,\nu,\mu,s) \xrightarrow{t}_\varepsilon (l,\nu',\mu',s)} \quad \texttt{Rule}_{Flow}$$

Rectangular Automata with Random Clocks

To enable reachability analysis in RAE, we need to lift the random events to the syntactical level. We do so by introducing a continuous variable for each random event. We refer to them as random clocks. Each random clock is assigned a continuous distribution. as was done for random events in RAE. To allow such a syntactical lift, we must ensure that the automaton is *stochastically non-guarded*. A RAE is *stochastically non-guarded* if the guards of all stochastic jumps equal $\mathbb{R}^d$. This ensures the random clocks valuation in Rectangular Automata with Random Clocks (RAC) is only defined by the Flow function of the location $Flow_R : Loc \to \{0,1\}^{d_R}$. For all $l \in Loc$ and $r \in Lab$, $Flow_R(l)_{|r} = 1$ iff there is a jump $j \in Jump$ with $source(j) = r$ and $Event(j) = r$, otherwise $Flow_R(l)_{|r} = 0$. Now that the rates of the random clocks are constant, they can be lifted to the syntactical level. When a random clock reaches the sampled expiration time $s_{|r}$ the according jump with $Event(j) = r$ must be taken. An additional clock variable t is used to measure the duration of a run [DSÁR23, DRÁ+25].

Definition 2.3.3 (Syntax of Rectangular Automata with Random Clocks [DSÁR23, DRÁ+25])**.** A Rectangular Automata with Random Clocks is a stochastically non-guarded RAE *H = (R, Lab,Distr,Event)* with *R = (Loc,Var,Inv,Init,Flow,Jump)* as defined for RA.

- Lab $\subseteq$ Var, with $\nu_{|r} = \mu_{|r}$ for each reachable state (l,ν,μ,s) of H
- t $\in Var \setminus Lab$ with $Init_{|t} = 0, Flow(l)_{|t} = [1,1], Inv(l)_{|t} = \mathbb{R}$ and $reset(e)_{|t} = id$ for all $l \in Loc$ and $e \in Jump$

The main differences between modeling a system as a RAE compared to a RAC, are that for a RAC the model designer has to ensure that no guards may be added to the stochastic jumps and that the dynamics of the random clocks corresponds to the stochastic jumps. While this adds modeling complexity, they are necessary to enable reachability analysis of RAE [DRÁ+25].

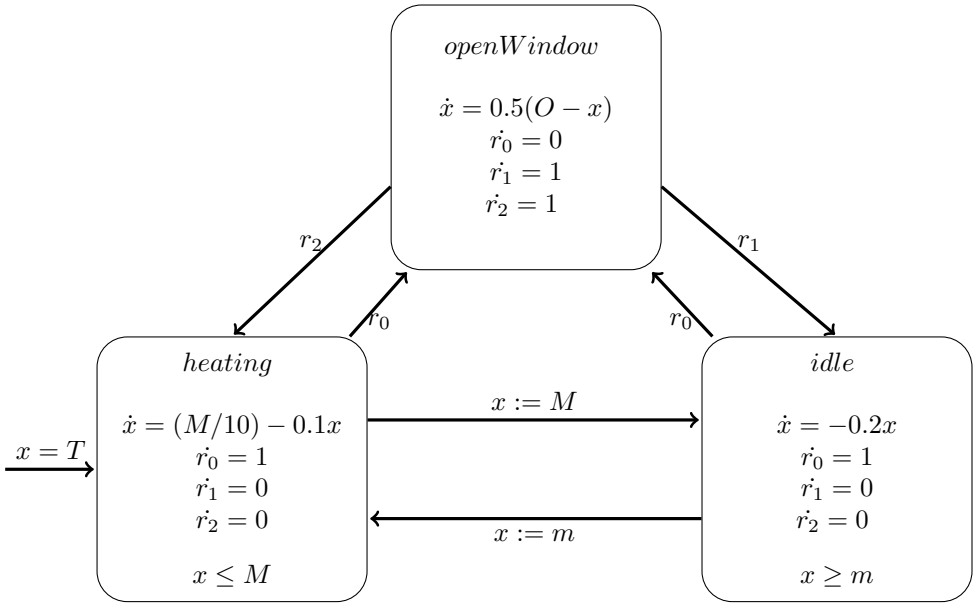


Figure 2.5: SimpleSLHA modeling a thermostat, adjusted from Figure 2.1. M, m, O and T represent the maximum, minimum, outside and room temperature, respectively. The timer variable t is omitted for notional simplicity.

2.3.2 Stochastic Linear Hybrid Automaton

We apply the same modeling approach to LHA. As with RACs we include random events syntactically by modeling random clocks to which we assign a sampling distribution. The syntactical definition is a refinement of RAC in Definition 2.3.3.

Definition 2.3.4 (Syntactical definition of a Stochastic Linear Hybrid Automaton)**.** A Stochastic Linear Hybrid Automaton is a tupel *H = (L,Lab,Distr,Event)* with *L = (Loc,Var,Inv,Init,Flow,Jump)* a LHA, and *Lab*, *Distr* and *Event* as defined in Definition 2.3.3 except for

- t $\in Var \setminus Lab$ with $Init_{|t} = 0$, $Flow(l)_t$=1, $Init_{|t} = \mathbb{R}$ and $reset(e)_{|t} = id$ for all l $\in$ Loc and e $\in Jump$

Figure 2.5 shows a Stochastic Linear Hybrid Automaton (SLHA) modeling the aforementioned thermostat with the uncertainty of an open window. We assume that the heating device can detect an open window and stops heating while it is open.

As was the case in RAC, the automaton must be stochastically non-guarded and that the dynamics of the random clocks must be set correctly such that $Flow_R(l) = \{0,1\}^{d_R}$ for all $l \in Loc$ with $Flow_R$ defining the flow of the random clocks. For each random clock r it must hold that if and only if there is a jump $j \in Jump$ with $source(j) = l$ and $Event(j) = r$, then $Flow_R(l)|_r = 1$. If no such jump exists $Flow_R(l)|_r = 0$.

2.4 Reachability Analysis

Reachability analysis for HA is a well-know safety measure for the modeled system. We can model undesired or desired states in the system as a set of of bad or goal states in the automata, also referred to as *specification*. Reachability analysis for HA tries to compute whether the defined specification is reachable from any state in the initial set.
Computing reachability in HA is generally undecidable. However, for certain classes, such as RA or LHA with linear behavior, bounded reachability is decidable. Bounded reachability is a subclass of reachability in which the automata execution is bound by time and, optionally, jump depth. LHA with nonlinear behavior are undecidable even for bounded reachability. Therefore, rather than computing the exact reachable sets, we aim to approximate the reachability. By computing the overapproximation of the reachable set, we ensure a safe system. This approach returns all reachable sets plus some overapproximation error. Thus, if a bad state is reachable, it is contained in the result of the overapproximation. If the intersection of bad states and reachable states determined by the overapproximation is empty, we know that the bad states are never reached in an exact computation. Therefore, the system is safe [SÁE22, Sch19]. However, if the overapproximation returns, that its intersection with the bad states is nonempty, we cannot conclude that the same applies to the exact result. While the underapproximation of reachable states has been given less attention in research, its impact should not be diminished, as it can be used to prove unsafety when given an undesirable state. The result computed with this approach is a set of guaranteed reachable states. If the intersection of the computed states and the bad states is nonempty, we can be certain that our system can enter a bad state [XSE16].

2.4.1 Flowpipe Construction

One popular approach to approximating reachability in LHA are flowpipe-construction based methods. As visualized in Figure 2.6, this approach takes, a geometric representation of the initial state set of a HA and iteratively computes that successor states. The values of variables in the automata develop over a fixed time horizon in defined time steps. The flow of a location is defined by a system of linear ODEs.

$$\dot{x} = \frac{dx}{dt} = Ax \tag{2.2}$$

with $x = (x_0,...,x_{d-1})^T$ representing the variables of the automaton, and $A \in \mathbb{R}^{d\times d}$. To solve Equation 2.2, we use the matrix exponential

$$x(t) = e^{tA} \cdot x(0)$$

to determine the state $x(t)$ reached at time point t from initial state $x(0)$ at time point 0. With Equation 2.4.1, it is possible to compute the reachable state at a specified time point t. However, it still provides insufficient information about the reachable states over the time interval. To resolve this issue, methods to approximate the reachable states for time intervals have been developed [Sch19].

Overapproximative Flowpipe Construction

Commonly, flowpipe construction is used to compute an overapproximation of the reachable states. As such, we aim to compute an overapproximation of the error

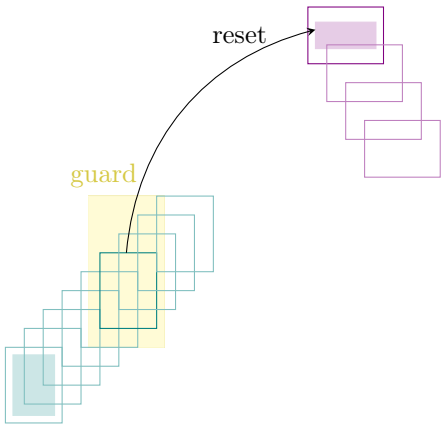


Figure 2.6: Flowpipe construction following a segment starting in one location. A jump to another location is performed for a time successor of the segment that fulfills the jump guard.

between the approximation of the reachable state set and the actual reachable state set in a given time interval. One approach to overapproximative forward flowpipe construction to LHA is to utilize the Minkowski Sum to initially compute a first segment [Sch19].

Definition 2.4.1 (Minkowski Sum [Sch19])**.** The Minkowski sum of two sets A,$B \subseteq \mathbb{R}^d$ is defined as the set

$$A \oplus B = \{a + b \mid a \in A \wedge b \in B\}$$

To compute the first segment Ω_0, the convex hull of the initial set at $t = 0$ and the state set after one time step $\delta : e^{\delta A}x(0)$ and bloat it using the Minkowski Sum. Figure 2.7 shows a sketch of the first segment computed from an initial set and its time successor after time $t = \delta$. With the same time step δ, we compute the successors of the first segment as

$$\Omega_{i+1} = e^{\delta A} \cdot \Omega_i$$

in one location l with $i <= \frac{T}{\delta}$ with T as the global time limit of the automata run when starting from $t = 0$. The computed segments must satisfy the invariant of location l. If the invariant is violated, no more time successors are computed in l. For each computed segment in l, we then need to check for each outgoing transition $e = (l,g,r,l') \in Edge$ whether the segment at least partially satisfies the guard g. For every segment this holds for, we use the intersection of the segment and g to compute the jump successor using the affine transformation defined in the reset function r. We check the intersection of the jump successor and the invariant of l' for non-emptiness. If an intersection is found to be non-empty, it is declared as a valid jump successor. With the resulting valid jump successors, we repeat the procedure starting by computing the first segment of l' as described. The analysis finishes when the global time limit T is reached or no valid segments are left [Sch19].

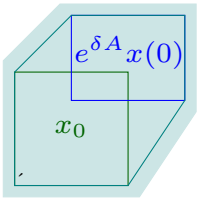


Figure 2.7: Sketch of first segment using bloating using using Minkowski Sum.

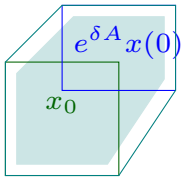


Figure 2.8: Sketch of first segment with debloating using Minkowski Difference.

Underapproximative Flowpipe Construction

Flowpipe construction for underapproximative reachability analysis is less explored. One method for it, is to replace the Minkowski sum with the Minkowski difference in the computation of the forward segment to debloat the convex hull instead of bloating it.

Definition 2.4.2 (Minkowski Difference [WKA24])**.** The Minkowski difference $A \ominus B$ of two sets $A,B \subseteq \mathbb{R}^d$ is defined as the set

$$\begin{aligned} A \ominus B &= \{c \mid c \oplus B \subseteq A\} \\ &= \{c \mid \forall b \in B : c + b \in A\} \end{aligned}$$

Figure 2.8 shows a sketch of a first segment that is computed using the Minkowski Difference. Except for the computation of the first segment in each location, underapproximative flowpipe construction follows the same steps as the overapproximative approach.

2.4.2 Reach Tree

With the reachability analysis, we can compute the reachable states of the automata. The flowpipe construction introduced previously computes a set of reachable segments. Those segments and their locations are stored in a reach tree. We define a *state set* (l,V) with V as a set of valuations where ν_x refers to the valuation of variable x in ν. Each flowpipe can be represented as a state set.

Definition 2.4.3 (Reach tree [DSÁR23])**.** For a LHA L, a time bound T, and time step t, a reach tree is defined as a tuple $R = (N,E)$ with a set of nodes $N \subseteq \mathbb{N} \times Loc \times 2^{\mathbb{R}^d}$ of the reach tree and edges $E = (N \times Jump \times N)$ with

- $n_i = (id_i,l_i,\{V_0^i,...,V_b^i\})$ with $b \leq \frac{T}{t}$ and $\{V_0^i,..V_b^i\}$ as the segments computed during forward flowpipe construction as consecutive time successors within l.
- for each $n_1 = (id_1,l_1,\{V_0^1,...,V_{b_1}^1\})$ and $n_2 = (id_2,l_2,\{V_0^2,...,V_{b_2}^2\})$ it must hold that $id_1 \neq id_2$, and if $(n_1,j,n_2) \in E$ then $id_1 < id_2$ for all $j \in Jump$, V_0^2 is jump successor of $V_{b_1}^1$ using jump j and $b_1 + b_2 \leq \frac{T}{t}$.
- a root node $root = (id_0,l_0,\{V_0^0,..V_b^0\}) \in N$ with $Init(l_0) \neq 0$, and $root \neq n_2 : \forall (n_1,j,n_2) \in E$.

- for every $n_2 \in N \setminus \{root\}$ there exists a unique $parent(n_2) = n_1 \in N$ with $(n_1,j,n_2) \in E$ for some $j \in Jump$
- $children(n_1) = \{n_2 \in N \mid \exists j \in Jump.(n_1,e,n_2) \in E\}$.

2.5 Non-Determinism

There are multiple sources of non-determinism in LHA and SLHA.
Discrete non-determinism: When multiple jumps are enabled at the same time, there is no natural choice of which jump to take, therefore not deterministic resolution of the situation [WRÁ23].
Continuous non-determinism: This source of non-determinism refers to the point in time when a jump is taken. If a jump is continuously enabled over a period of time, there is no semantic indication of when the jump is taken. However, different times of taking the jump may result in different outcomes of the automaton run [WRÁ25].
Initial-state non-determinism: This type of non-determinism occurs from the caveat that the initial state of an automaton may be chosen from a defined set of possible initial states. The choice of the initial state may influence the outcome of an automaton run [DSÁR23].

In this thesis, we investigate the issue of initial-state non-determinism and propose an approach to solve it probabilistically.

Chapter 3

Method

In this chapter, we will introduce the method to compute the backwards refinement in SLHA to solve initial-state non-determinism. As explained in Chapter 2, initial-state non-determinism occurs, when the initial state must be selected from an initial set of states. Figure 3.1 shows a SLHA reflecting a marathon. A runner starts off with a certain level of energy and a certain speed, having not passed any distance. Therefore, the initial state for energy is defined to be within the interval [1,5], the speed within [10,20], and the distance as well as the global timer are initialized as 0. Stochastic variables are always initialized to 0. If the runner runs out of energy, she passes out. The runner can take breaks at random points in time to replenish energy.

As introduced in Definition 2.3.4, we define a probability distribution for the random clocks in the automaton. To enable backwards refinement to determine the probability that the goal set can be reached from a state chosen from the initial set, we also define probability distributions over the continuous variables in the initial set. For the marathon-automaton in Figure 3.1, we define a probability distribution for *energy* to sample a value between [1,5] and for *speed* between [10,20]. Section 3.3 explains how the probability distributions are utilized in the analysis after backwards refinement. Forward flowpipe construction must be performed first to enable backwards refinement.

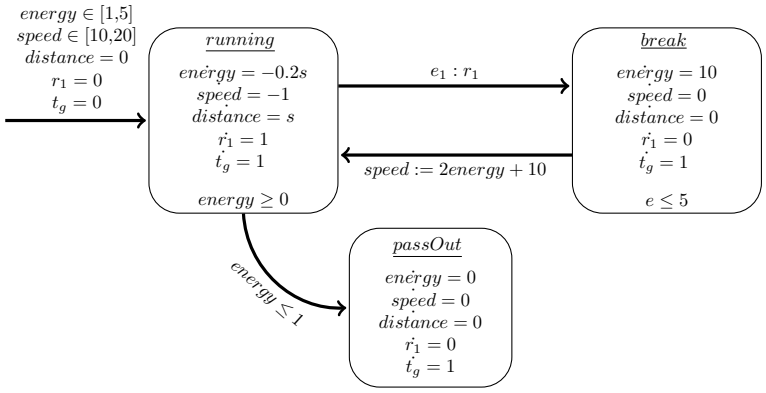


Figure 3.1: SLHA to model a marathon run. The runner can take breaks at random points in time recharge so her energy does not run out.

3.1 Forwards Reachability

In Chapter 2, we introduced forward flowpipe construction. Starting from the initial set with a given a time and jump bound, the forward reachable states are approximated. Figure 3.2 visualizes forward flowpipes for a two-dimensional automaton[1]. In the first location, two forward time steps are performed. Therefore, for each timestep the segment is calculated based on the previous segment and the location's flow. Then, a jump to the next location is performed. According to the jump's reset, the post-jump segment is computed. With the first segment in the new location, three forward time successors are computed. The forwards analysis returns a reach tree containing the root node in l_0 with a child in l_1.

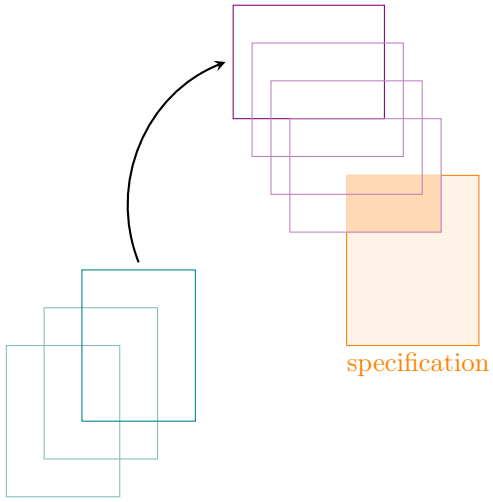
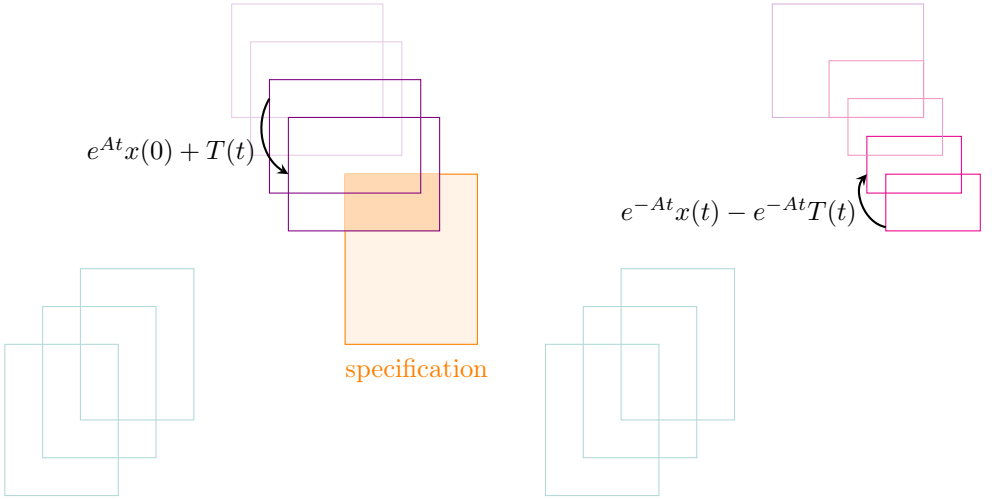


Figure 3.2: Forward flowpipes computed from a segment following two timesteps in one location before taking a jump with a reset and performing 3 more timesteps.

3.2 Backwards Refinement

We utilize backwards refinement to extend the reachability analysis, and thereby to determine which subset of the initial set can lead to a state in the specification. For backwards refinement, we examine the computed reach tree for flowpipes that intersect with the goal states, i.e., those flowpipes whose state space overlaps with the state space of the goal states. We use the terms goal states and specification analogously. In Figure 3.2, the specification overlaps with two of the flowpipes in the second location. We store those intersections between the flowpipes and the goal states – here referred to as segments or goal-satisfying segments – and use them as the entry points for the backwards refinement. For each segment, we iterate through two steps:
1) Follow each segment back to the initial set of the current node in the reach tree by

[1]The computation of the first segment is simplified in the figures in this chapter. We omit discussing the differences between the first segment and the initial set in this chapter but defer this to Chapter 5.



(a) Sketch of the forwards time step of a segment.

(b) Sketch of the backwards time step of a segment.

Figure 3.3: Sketch of forwards and backwards time successor computation.

computing backwards time transformations, then
2) Perform a backward jump to reach the parent node in the reach tree.
These steps are iterated until the reach tree has been reversed and the root is reached where no backwards jump can be performed. The final segments signify the subset of the initial set that led to the originally stored segments.

3.2.1 Backwards Time Computation

Each location of the automaton is assigned a flow to define the dynamics of the variables. A flow is defined by a system of linear ODEs. The flow function is used to compute the time successors during forward flowpipe construction. In Figure 3.3a the last two flowpipes intersect with the goal set. We regard the backwards refinement of the intersection of the last flowpipe with the goal set. We calculate back to the initial set of the node by performing backwards time steps. Therefore, we must reverse the application of the affine transformation, i.e. we must apply the inverse flow to the segment. The flow is defined by

$$\dot{x} = Ax + b \tag{3.1}$$

The time successor is computed for a fixed time step t. As introduced in Chapter 2, we can use the matrix exponential to solve for the time successor. Equation 3.1 can be transformed into its homogeneous form of

$$\frac{d}{dt}\begin{bmatrix} x \\ 1 \end{bmatrix} = \begin{bmatrix} A & b \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x \\ 1 \end{bmatrix}$$

If we define

$$y = \begin{bmatrix} x \\ 1 \end{bmatrix}, M = \begin{bmatrix} A & b \\ 0 & 0 \end{bmatrix}$$

this results in

$$\frac{dy}{dt} = My$$

Solving the ODE requires forming the matrix exponential.

$$y(t) = e^{Mt}y(0)$$

which is equivalent to

$$\begin{bmatrix} x(t) \\ 1 \end{bmatrix} = e^{Mt} \begin{bmatrix} x(0) \\ 1 \end{bmatrix}$$

Structurally, the matrix exponential is of the form

$$e^{Mt} = \begin{bmatrix} e^{At} & T(t) \\ 0 & 1 \end{bmatrix}$$

$T(t)$ is the translation vector defined as

$$T(t) = \int_0^t e^{As} b \, ds$$

We can restructure the formula as

$$x(t) = e^{At}x(0) + T(t)$$

Inverting that representation results in

$$x(0) = (e^{At})^{-1}x(t) - (e^{At})^{-1}T(t) = e^{-At}x(t) - e^{-At}T(t)$$

With this equation, the backwards time successor can be computed [Sch19].

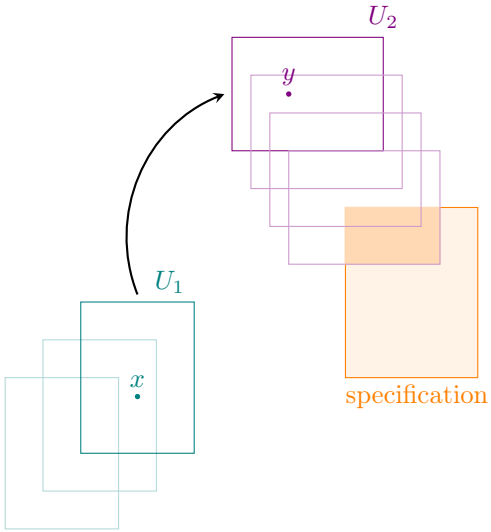
As shown in Figure 3.3b, the reverse flow is applied to the segment to compute the backwards time segment. We repeat computing the backwards time successors until the first segment of the location is reached.

3.2.2 Backwards Jump

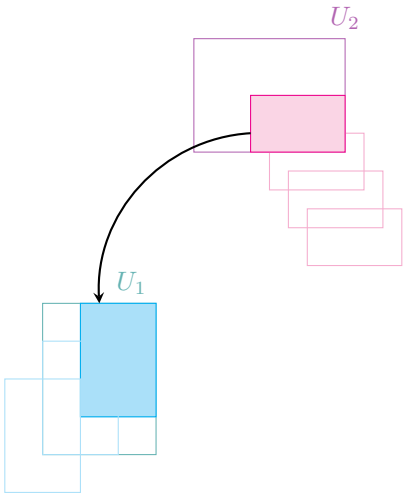
Between the locations of the automaton, jumps are performed to get from one location to the next one. A jump in an automaton can be enhanced with resets and guards as described in Chapter 2. As shown in Figure 3.4a, during a jump reset, the values of the variables change according to the defined transformation. In the backwards refinement, the transformation has to be reversed to reach the pre-jump segment.

Backwards Reset

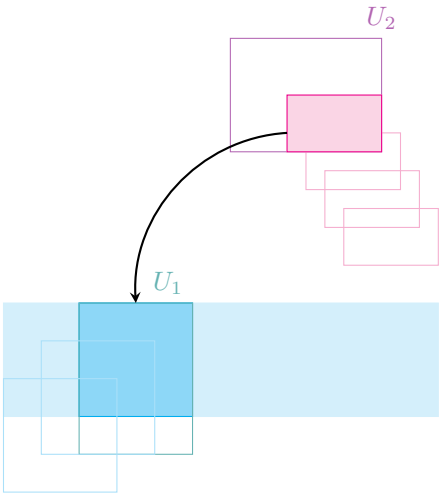
A *jump reset* is defined as a linear transformation $Ax + b$. However, we cannot compute the pre-jump segment as we did for the backwards time successor. Firstly, the computation of the reset is not defined continuously over some time step. Thus, no matrix exponential is formed as we did for the time successor and we cannot guarantee that A in the reset $Ax + b$ is invertible. Secondly, when we computed the backwards time successor, we could rely on the fact that the value of each variable was uniquely defined by the flow of the location. This guarantee is not provided by the reset. The valuation of a variable after the reset may be independent of its previous value. Take



(a) U_1 represents the segment before the reset and U_2 represents the segment after the reset. y is a point in U_2 and x is a points in U_1.



(b) Backwards reset to compute pre-jump segment with all dimensions uniquely defined.



(c) Backwards reset to compute pre-jump segment with one dimensions undefined dimensions. Pre-jump segment must be intersected with the forwards segment U_1 to limit the valuation of the dimension.

Figure 3.4: Sketch of a forwards and backwards jump including a reset.

e.g. a reset where $x = 5$. We know that after the jump, the value of x is always 5 no matter the previous value. Therefore, we cannot limit the previous value of x by applying the reverse reset. Instead, we apply a different algorithm, that depends on HPolytope representation, provided by Hypro, of the segments.
An HPolytope H can be defined with a matrix A and a vector b as

$$H = \{x \in \mathbb{R}^n | Ax \leq b\}, A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$$

We exploit this representation to compute the pre-jump segment. The pre-jump segment U_1 and the post-jump segment U_2 are defined as HPoyltopes, as shown in Figure 3.4a.

$$U_1 = \{x \in \mathbb{R}^n | K_1 x \leq c_1\}, K_1 \in \mathbb{R}^{n \times n}, c_1 \in \mathbb{R}^n$$
$$U_2 = \{y \in \mathbb{R}^n | K_2 y \leq c_2\}, K_2 \in \mathbb{R}^{n \times n}, c_2 \in \mathbb{R}^n$$

For the jump between U_1 and U_2, the reset $Ax + b$ is defined

$$U_2 = AU_1 + b, A \in \mathbb{R}^{n \times n}, b \in \mathbb{R}^n$$

For every point in U_1, there is a point in U_2 that is computed by applying the reset function. In Figure 3.4a, the point y in U_2 is computed by applying the reset function $y = Ax + b$ to x in U_1.

$$\forall y \in U_2 \; \exists x \in U_1 : y = Ax + b$$

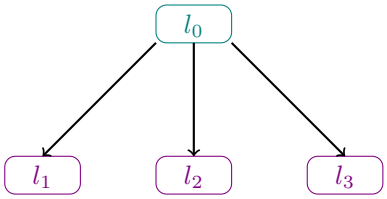
In the HPolytope definition of U_2, we can replace y with the reset function

$$K_2(Ax + b) \leq c_2$$
$$K_2 Ax \leq c_2 - K_2 b : \forall x \in U_1$$

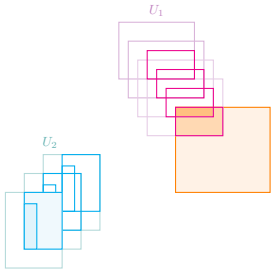
which provides a definition of U_1 using the available information about U_2 and the reset function.

$$U_1 = \{x | K_2 Ax \leq c_2 - K_2 b\}$$

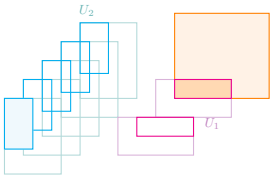
As shown in Figure 3.4c, we can apply this transformation to the backwards refined segment that was computed through backwards time steps, to receive the pre-jump segment. However, as mentioned above, there may be dimensions that were not referenced in the reset which results in a pre-jump segment that still contains infinite dimensions as visualized in 3.4b. To limit the infinite dimensions to the correct values, the calculated segment must be intersected with the forward segment. The forward segment is the segment in the trace that we took the jump from in the forward flowpipe construction – U_1 in our visualization. This forward segment, therefore, needs to be known. The specifics on how to get the forward segment depend on the implementation. The details of the approach utilized in this thesis are given in Chapter 5.



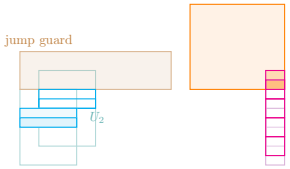
(a) Reachtree for the automaton containing traces in Figure 3.5b- 3.5d.



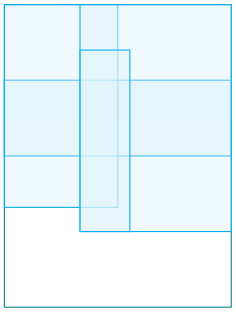
(b) Forward flowpipes from l_0 to l_1 with backwards refinement of two segments that intersect with the specification.



(c) Forward flowpipes from l_0 to l_2 with backwards refinement of two segments that intersect with the specification.



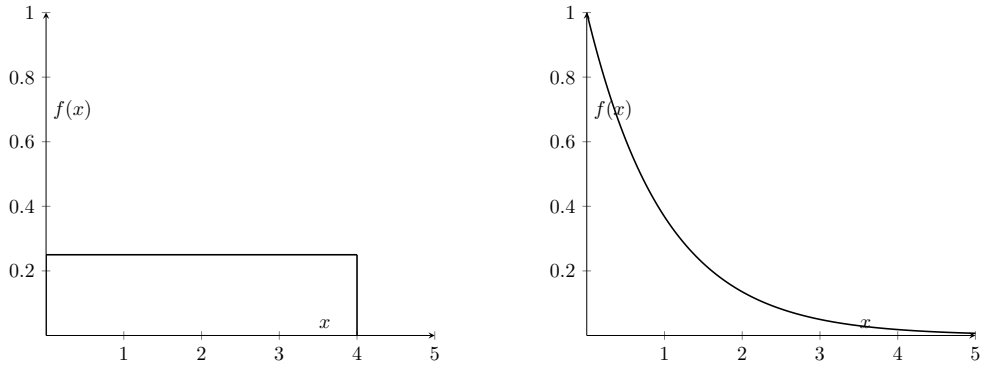
(d) Forward flowpipes from l_0 to l_3 with backwards refinement of two segments that intersect with the specification.



(e) Combination of backwards refined segments.

Figure 3.5: Sketch of backwards refinement following different paths in the reach tree.

With these steps, the backwards time successors and backwards jumps of the goal-satisfying segments are computed until the time spent going forwards has been gone backwards. Therefore, we traced backwards to the subset of the initial state set that resulted in the goal-satisfying segment. Once, all segments have been processed, we know the total subset of the initial set that can reach a goal-satisfying state. As an example, we regard a SLHA with the reach tree shown in Figure 3.5a. From the initial state l_0, three locations l_1, l_2, l_3 can be reached. Figure 3.5b shows one trace following the path from l_0 to l_1. Two of the forward flowpipes intersect with the specification, thus we backwards refine those two segments until the initial set of l_0 is reached. Figure 3.5c similarly shows a trace that follows the path from l_0 to l_2, with one flowpipe intersecting with the specification, and Figure 3.5d equivalently for l_3 with two flowpipes intersecting with the specification. This results in a total of five



(a) Uniform distribution for x values in interval [0,4].

(b) Exponential distribution for x values in interval [0,4].

Figure 3.6: Examples for continuous distributions that can be used for the variables initial set.

segments that were backwards refined and thus five resulting segments, each a subset of the initial set. In Figure 3.5e all result segments are shown within the the initial set of the root node l_0.

3.3 Probability Computation

In the final step, we process the computed segments to determine the probability that any run starting in the initial set succeeds given the defined time and jump depth. It is important to note that the probability calculated in this step does not account for other types of non-determinism that can occur. A run starting in an initial state that is within the computed bounds may still be unsuccessful if unsuitable expiration times for random clocks are used. To account for all types of non-determinism that can be resolved probabilistically, the different sources have to compute their success probability individually, and the results must be combined. More insight into this aspect is given in Chapter 7.

The values of variables in the initial state set were defined over a distribution. Depending on the kind of distribution, different points in the interval are chosen with a different probability. For example, with uniform distribution (Figure 3.6a all values on the interval have the same probability of being sampled, while for an exponential distribution (Figure 3.6b) lower values on the interval have a higher probability to be selected. To ensure that all types of probability distributions are handled correctly, we use Monte-Carlo Integration over initial values [Pet78]. GLS provides the Monte Carlo integration. We use the computed polytopes that intersect the initial state set and project out all dimensions that were assigned a singular initial value from the probability computation. This implicitly also projects out all stochastic dimensions which are initialized with zero. We use the intervals defined in the initial set as the integration bounds and integrate statistically using Monte-Carlo Integration.

Chapter 4

Tools

In this thesis, we use and extend two main frameworks: Hypro and Realyst. This work extends the existing and currently ongoing work. These methods span over Hypro and Realyst, which are introduced in the following sections.

4.1 Hypro

Hypro[1] is a project developed mainly at RWTH Aachen. It is a toolbox providing methods for reachability analysis of HSs with mixed discrete-continuous behavior and provides multiple implementations of state set representations for reachability analysis.

The reachability analysis provided by Hypro uses flowpipe construction based methods to apply to time-bound systems to compute reachable states within this time bound. There are several datatypes available to represent the state sets of HSs. In this thesis, polytopes are used for the state set representation.
In recent years, work has been invested to extend Hypro for the analysis of LHA. Overapproximative methods for forward flowpipe construction have already been successfully implemented. Other ongoing work focuses on underapproximative flowpipe construction to eventually be able to provide an upper and lower bound for reachability in LHA. This work extends the existing and currently ongoing work by providing a method for backwards time computation in (S)LHA [SÁMK17].

4.2 Realyst

The Realyst project[2] is currently under development at RWTH Aachen and WWU Münster as a collaborative undertaking. Realyst is an open-source C++ tool for computing time-bound reachability in Stochastic Hybrid Automata. It uses flowpipe construction, provided by the Hypro toolbox and GLS[3] for multi-dimensional integration. Realyst uses convex polytopes as the state set representation. As our

[1] https://ths.rwth-aachen.de/research/projects/hypro/

[2] https://ths.rwth-aachen.de/research/projects/realyst-reachability-analysis-for-stochastic-hybrid-systems/

[3] https://www.gnu.org/software/gsl/

work extends the Realyst Toolset, we also rely on convex polytopes as the state set representation [DSSR24].

4.3 Contribution

We contribute to the Hypro framework by implementing a method to compute backwards time successors in (S)LHA as is described in Chapter 3. Realyst is extended by solving non-determinism introduced by the initial set probabilistically through backwards refinement.

Chapter 5

Implementation

In this chapter, we go into more detail about how the concepts that were introduced in Chapter 3, have been implemented in Realyst and Hypro. We first go through the existing foundation that computes forwards reachability which builds the base for the implementation of our proposed method. Then, we explain how backwards reachability analysis for the purpose of resolving initial state non-determinism probabilistically is realized. The complete analysis is organized in five steps, as listed in Figure 5.1. Each step is inspected in the following. In Figure 5.2 relevant classes and their attributes from Realyst and Hypro are presented.

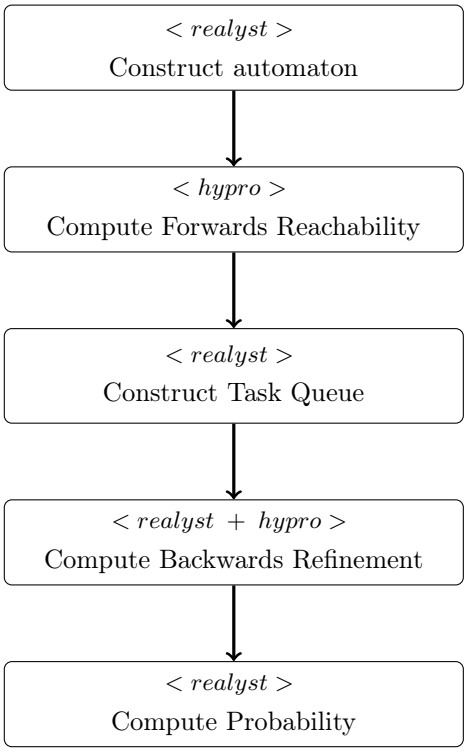


Figure 5.1: Steps for backwards refinement analysis.

5.1 Construct Automaton

The user has to define an automaton in Realyst and must ensure syntactical correctness. Otherwise, the construction of the automaton or, later on, the reachability computation will fail.
The user must define:

- The continuous and stochastic variables
- The locations
- The flow in each location
- The invariants in each location
- The jumps between the locations
- The guard of each jump
- The reset of each jump
- The probability distribution over the stochastic variables
- The probability distribution over the initial set
- The possible initial values for every variable
- The goal specification

If no flow is defined for a location, we assume that the values of the variables do not change within the location. If no reset is defined for a jump, we assume that the values of the variables are not impacted when the jump is performed. Invariants and guards may be empty. In order to solve initial state non-determinism probabilistically, a probability distribution over each variable in the initial set must be given. The distribution must only be defined for variables, whose value is not defined by a single value. This implicitly excludes random clock that all have an initial state of 0. If no initial state probabilities are defined, Realyst extracts the probability distribution automatically as a uniform distribution over all possible values for each variable in initial set that is defined over an interval.

5.2 Compute Forward Reachability

The computation of the forward reachability is implemented in Hypro and called from Realyst. First, the roots of the reach tree are determined which serve as the foundation to further compute forward reachability. Hypro implements forward flowpipe construction as introduced in Section 2.4.1. Both an over- or underapproximative method are available. The algorithm computes the automaton's reach tree. A reach tree is built of reach tree nodes. As shown in Figure 5.2, an object of class `ReachTreeNode` contains the attributes:
location: The `location` attribute stores the location of the automaton as an object of the `Location` class. As such it stores the location's flow, invariant and outgoing transitions.
transition: The `transition` attribute stores the incoming transition i.e. the transition from the parent to this ReachTreeNode. An object of the `Transition` class stores the source and the target location, as well as the guard and the reset of the

associated jump.
flowpipes(UnderApprox): This attribute stores the consecutively forward flowpipes computed during forward reachability analysis in the location of the node. `flowpipes` stores the overapproximative flowpipes and `flowpipesUnderApprox` the underapproximative flowpipes.
initialSet(UnderApprox): This attribute contains the initial set of the location, i.e. segment right after the jump to this current location was taken. With the initial set, the first segment in the `flowpipes` vector is calculated.
timings: The timings attribute stores the global time interval that is covered by the initialSet of the location. For the root of the reach tree, the timings of the initial set is [0,0]. We build the first segment from the initial set and its first time successor. Therefore, the firstSegment of the root node covers the global time [0,1]. We then calculate n time successors of the first segment before jumping to the next location. The initial set of child node then covers the global time [n,n+1]. With every jump the interval covered by the initial set of the location is thereby increased by 1. Figure 5.4 shows how the first segment spans more time with each jump.

The reach tree is processed to determine forward reachability by checking if any of the computed flowpipes intersect with the specification. If so, the goal is reachable and the program can continue with backwards refinement. Otherwise, if the goal is not reachable, the computation is finished with an initial-state probability of zero.

5.3 Construct Task Queue

The reach tree is used to computed a task queue. The task queue represents the segments that need to be backwards refined. To build the task queue, every node in the reach tree processed to check which of the stored flowpipes at least partially intersects with the specification. Each intersection is stored as a segment in a `TraceElement`. The TraceElement further stores the according node in the reach tree, a segmentType and a segmentIndex. Initially, all segments are assigned the type `BACKWARDS_INTERMEDIATE_GOAL_SEGMENT`. The segmentIndex stores the position of the original flowpipe in the *flowpipes* vector. The TraceElements of one node are stored in a vector. We alias a vector of TraceElements as `Trace`. All Traces are stored in a vector, which is kept as the attribute `traces` in a `TraceRepository`. For each node with at least one intersecting flowpipe as `Task` object is created. It stores the (Stochastic)ReachTreeNode, a traceIndex and a segmentIndex. The traceIndex refers to the index of the Trace associated with the node in `traces`. The segmentIndex stores the size of the Trace - 1, therefore the index of the last TraceElement in the Trace.

In Figure 5.3, five flowpipes from two location intersect with the specification. From this, a TraceRepository with two Traces would be created. One Trace contains three TraceElements with the node in location l_1, and the other Trace two TraceElements with the node in location l_2. The segmentIndex of each TraceElement stores the value i indicated in the figure. Two Tasks are created, one for node with location l_1 with traceIndex 0 and segmentIndex 2 and one for the node with location l_2 with traceIndex 1 and segmentIndex 1.

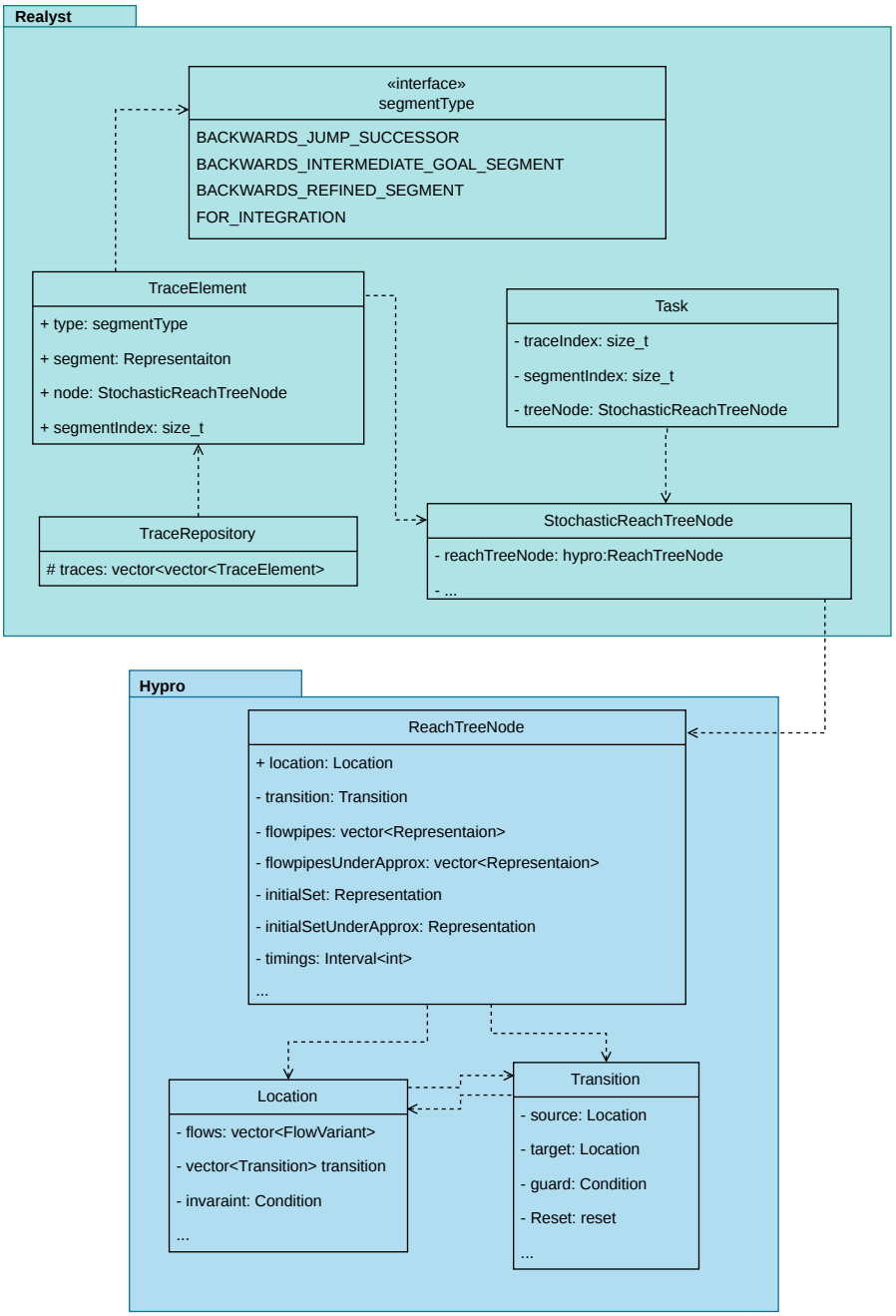


Figure 5.2: Simplified class diagram of relevant classes in Realyst and Hypro.

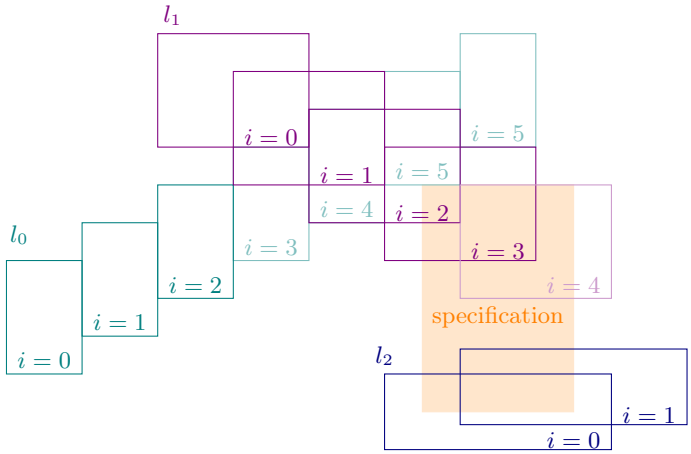


Figure 5.3: Forward flowpipes of three nodes with two jump and time bound 6.

5.4 Backwards Refinement

To compute the backwards reachability, the task queue is iterated and each trace is processed. Algorithm 1 shows the pseudo code of *initialStateAnalysis* which implements the foundation of this step. In this method, the task queue is iterated and each task is processed depending on the the *segmentType* of the segments in the trace associated with the current task.

5.4.1 Backwards Time Step

First, we regard the case that segmentType is `BACKWARDS_INTERMEDIATE_GOAL_SEGMENT`. For segments of this type the backwards time successors must be computed. The method *computeBackwardsTimeRefinement* in Algorithm 2 is called on the trace. In this step, the backwardsTimeSuccessors are iteratively calculated as explained in Chapter 3. For each segment stored in a TraceElement, we call *applyBackwardsTimeEvolution* for the number of steps stored in *segmentIndex* of the TraceElement. The value is either set during task construction as explained or calculated in a later processing step. *ApplyBackwardsTimeEvolution* calculates one backward time step as explained in Chapter 3. Once all necessary backsteps have been performed, the resulting segment is intersected with the initialSet of the node. We do so for every TraceElement in the trace. For every TraceElement, the segmentType is then changed to `BACKWARDS_REFINED_SEGMENT`. Since the `switch case` is defined to fall through between cases, we move on with the same task and handle the next case.

A segment may become or be marked as empty during backwards time computation in two cases:

1. If the segment or one of its backwards time successors has very narrow dimensions or is only a point. Due to numerical restrictions of the underlying solver, very narrow segments may be marked as empty and can thus not be further processed.

2. When the backwards time successors are computed, the intersection of the backwards refined segment and the initial set can be empty. As discussed the initial set covers timings [n,m] and the forward flowpipe i covers [n+i,m+i+1]. During backwards refinement, i time predecessors of the intersecting segment are computed. If the intersection is the part of the original flowpipe covering the time [m+i,m+i+1], the backwards refinement of the segment will not overlap with the initial set of the location. This is not a issue and the valuations contained in the lost segment are not disregarded. If $m + i + 1 \leq T$, the segment i+1 will contain the intersection as well. The backwards refinement of that segment will overlap with the initial set of the location. The redundancy of this occurrence is discussed in Chapter 7.

5.4.2 Backwards Reset

In this step, the backwards jump is performed. Therefore, we first need to make sure that the current trace has a parent node that we can jump to. If there is no parent, we have reached the root node of the reach tree and all backsteps have been performed, the initial state is reached and the type of the TraceElement is changed to `FOR_INTEGRATION` marking the trace as ready for Monte-Carlo-Integration. If there is a parent node, `computeBackwardsJumpReset` is called, shown in Algorithm 3. In this method, the backwards jump is computed as explained in Chapter 3 using available information from the segment and the transition. To overcome the issue of possible infinite dimensions, the computed pre-jump segment must be intersected with the forwardSegment. The forwardSegment is the flowpipe from which the jump to the current node in the reach tree was taken. As shown in Figure 5.4, flowpipe in location l_1 with $i = 3$ is the forward segment for the nodes in location l_2. The ReachTreeNode do not store a reference to this segment but it can be computed using the `timings` attribute. The initialSet of node with location l_2 covers the global time [5,7], the initialSet of the node with location l_1 covers [2,4]. Therefore, we can compute that the wanted forwardSegment is found in the `flowpipes` vector of node with location l_1 at index $5 - 2 = 3$.

We compute the pre-jump segment for every TraceElement in the trace. Next, we update the TraceElement for the next processing iteration. All computed segments are positioned at the same point in time, therefore for all segments can use the same new segmentIndex, which is used as the number of necessary backwards time evolution step. We already computed the index of the forwardSegment. The segmentIndex stores the same value as the computed index. The forwardSegment is the flowpipe after i forward time steps from the firstSegment. This is the same number of backward timesteps necessary for the pre-jump segments.

After iterating through all tasks, the resulting segments are reduced to only include the dimensions of those variables whose initial state is defined over a distribution by projecting out the other dimensions. With the final `unionOfPolytopes`, we continue to calculate the initial state probability using Monte-Carlo Integration as described in Chapter 3

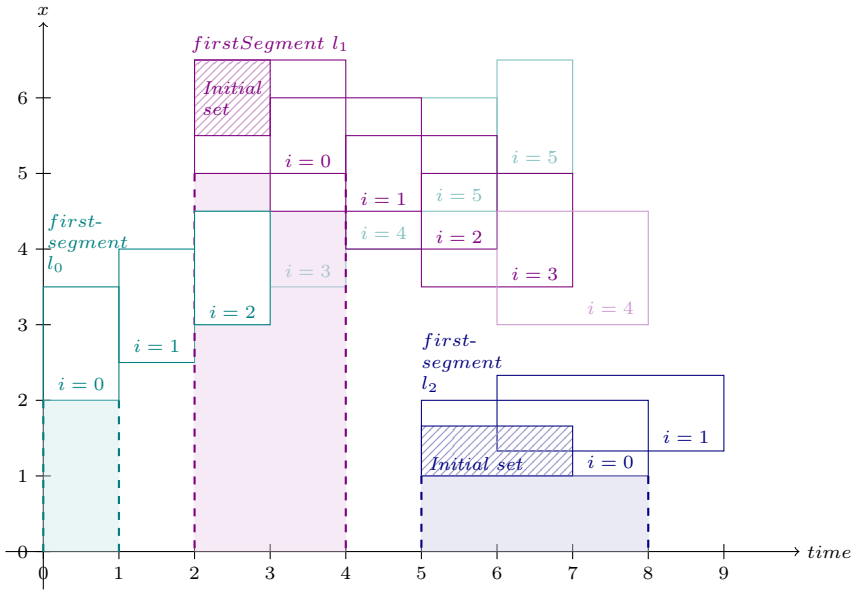


Figure 5.4: The x-axis shows the global time. The first segment of the first location covers the time [0,1] because it was built from the initial state with timing [0,0] and the first time step (simplified overapproximation). The trace stored for l_0 is not only computed until the jump but until a time limit or another stop condition is reached. The time limit is reached when the *lower* bound of the timings is equal to the maximum time. The first segment of l_1 covers the time [2,4] because the jump to it occurred at $time = 2$ and the first segment was formed of the post-jump segment with timings [2,3] and the next timestep.

Algorithm 1 initialStateAnalysis

```
Function initialStateAnalysis(tasks, settings, segments, spec, pool, poolPtr,
sha, initialDistributions, underApprox):
    while tasks not empty do
        current_task ← tasks.pop()
        node ← current_task.treeNode
        parent_node ← current_task.treeNode.parent
        trace ← segments.getTrace(current_task.traceIndex)
        traceElement ← trace[current_task.segmentIndex]
        switch traceElement.type do
            case BACKWARDS_INTERMEDIATE_GOAL_SEGMENT do
                computeBackwardsTimeRefinement(node, trace, current_task, settings)
                foreach elem ∈ trace do
                    elem.type ← BACKWARDS_REFINED_SEGMENT
                // Fall through
            case BACKWARDS_REFINED_SEGMENT do
                if parent_node ≠ NULL then
                    computeBackwardsJumpReset(current_task, trace, parent_node)
                    timings ← current_task.treeNode.reachTreeNode.getTimings()
                    prevTimings ← parent_node.getTimings()
                    newSegmentIndex ← timings.lower() − prevTimings.lower()
                    foreach elem ∈ trace do
                        elem.type ← BACKWARDS_INTERMEDIATE_GOAL_SEGMENT
                        elem.segmentIndex ← newSegmentIndex
                        elem.nodeptr ← parent_node
                    tasks.emplace(new Task(current_task.traceIndex, trace.size() - 1,
                    parent_node))
                else
                    initialSet ← current_task.treeNode.getInitialSet()
                    foreach t ∈ trace do
                        t.segment ← t.segment.intersect(initialSet)
                        t.type ← FOR_INTEGRATION
            case FOR_INTEGRATION do
                // Ready for integration, no further processing
            otherwise do
                throw error("Unknown task type")
    unionOfPolytopes ← findIntegrationBoundsForInitialSet(segments, sha)
    return unionOfPolytopes
```

Algorithm 2 computeBackwardsTimeRefinement

```
Function computeBackwardsTimeRefinement(node, trace, task, settings):
    index ← trace.size()
    traceIndex ← task.traceIndex
    initialSet ← node.getInitialSet()
    backSteps ← 0
    while index ≥ 0 do
        segment ← trace[index].segment
        segmentIndex ← trace[index].segmentIndex
        while backSteps < segmentIndex do
            segment ← applyBackwardsTimeEvolution(segment, node, settings.timeStep)
            backSteps ← backSteps + 1
        intersected ← segment.intersect(initialSet)
        trace[index] ← intersected
        index ← index − 1
    return trace
```

Algorithm 3 computeBackwardsJumpReset

```
Function computeBackwardsJumpReset(task, trace, parent_node):
    reset ← task.getReachtreeNode().getTransition().getReset()
    if reset.isIdentity() then
        return trace
    resetMatrix ← reset.getMatrix()
    resetVector ← reset.getVector()
    flowPipes ← parent_node.getFlowpipes()
    prevTime ← parent_node.getTimings().lower()
    currentTime ← task.getReachtreeNode().getTimings().lower()
    forwardSegmentIndex ← currentTime − prevTime
    forwardSegment ← flowPipes[forwardSegmentIndex]
    foreach traceElement ∈ trace do
        segment ← traceElement.segment
        segmentMatrix ← segment.matrix()
        segmentVector ← segment.vector()
        K ← segmentMatrix × resetMatrix
        f ← segmentVector − (segmentMatrix × resetVector)
        newSegment ← Polytope(K, f)
        newSegment ← newSegment.removeRedundancy()
        newSegment ← forwardSegment.intersect(newSegment)
        traceElement.segment ← newSegment
    return trace
```

Chapter 6

Results

In this chapter, we demonstrate the feasibility of our algorithm by providing the results of backwards refinement on benchmarks of various complexity. The resulting initial-sate probability, and the run-time of the backwards refinement are provided. In addition, we also discuss threats to the validity of our results.

6.1 Benchmarks

In this section, various benchmarks are explored. The automaton of each benchmark is backwards refined and the initial state probability is computed. We measure the time for forward analysis and backward analysis. Forward analysis includes the computation of over- and underapproximative flowpipes. We provide the resulting probability of the backwards refinement based on both types of forward analysis. The plots provided in this chapter reflect the overapproximative analysis if not otherwise specified. For smaller examples, we give an estimate of the expected result. For bigger examples, such an estimation by hand is not feasible. However, with the smaller examples, all properties of the proposed method are covered to give a sound assumption that the backward analysis for more complex benchmarks is correct. It is not possible to compare the results of the backwards analysis to the results of a different tool, since we are not aware of another tool that offers backwards reachability analysis for SLHA aiming to resolve initial-state non-determinism.
The commands to run the benchmarks in Realyst are given in Appendix A.

6.1.1 Simple Benchmarks

First, simple benchmarks are introduced, whose results can be estimated. In order to estimate the expected initial-state probability, we always take a uniform distribution over the initial values. Since the Monte-Carlo integration is provided by GLS, we can safely assume it to be correct and deliver correct results for different types of distributions as well.

Simple Case A

For the first benchmark, a very simple automaton is analyzed, displayed in Figure 6.1. This automaton only has two locations and one stochastic jump. Only variable

x has an initial value defined over an interval. In the reset, only the value of x is changed and is, hereby, dependent on its previous value.

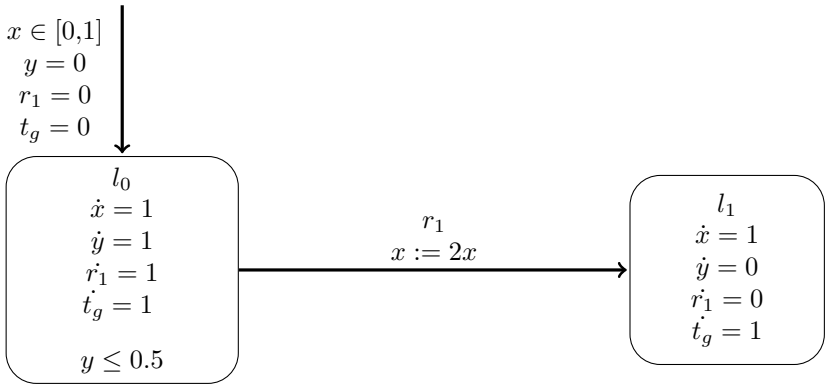


Figure 6.1: Simple SLHA with two location and a stochastic jump between them. There are three continuous variables x,v and a global timer t_g and one stochastic variable r_1.

For the example in Figure 6.1, we choose the specification:

$$x \geq 2$$
$$t_g \leq 1$$

To properly handle the global time limitation, we include it directly in the specification[1]. We define a uniform probability distribution over [0,1] for the stochastic variable r_1, as well as for the continuous variable x.

Estimate: Estimating a result for this benchmark is simple, because there is only one variable with a nondeterministic initial value. We aim to determine the smallest possible value for x to still reach the specification. Since x is only ever increased and never limited in value, every higher initial value will also reach the specification. We define x_{end} as the final value of x, x_{init} as the initial value of x with $x_{init} \in [0,1]$, T as the time bound and t_i as the time spent in the location i. After increasing with a flow of 1 in location l_0, the value of x is doubled when the jump to l_1 is taken. In l_1 the flow is defined as the constant 1 as well. This gives the formula

$$x_{end} = 2(x_{init} + t_o) + (T - t_0)$$

Due to the invariant of l_0 specify $y \leq 0.5$ and $\dot{y} = 1$ in l_0 with $y_{init} = 0$, we know that the maximum time spend in l_0 is 0.5. Since the value of x is double in the reset

[1]The result of the backwards refinement remains the same without including the global time in the specification. However, as the forward analysis stops when the lower bound of the covered time interval reaches the time bound, a specification that is only reached after the maximum time may be declared as reachable after forward analysis. By including the time bound in the specification, we avoid such a scenario. For backwards analysis, only segments within the time bound are considered, in either case.

of the jump, we want to spend as much time as possible in l_0, thus t_0 is fixed to 0.5. The total time T is set to 1 in the specification. According to the specification, x_{end} must be at least 2, resulting in the formula

$$\begin{aligned} 2 &\leq 2(x_{min} + 0.5) + (1 - 0.5) \\ x_{min} &\geq 0.25 \end{aligned}$$

With a uniform distribution, this should result in a probability of 0.75% as three-fourths of the initial set can lead to a goal-satisfying state. As listed in Table 6.1, our algorithm produces the same result. Plot 6.2 shows the computed backwards refined segments. The x-axis shows the value of the variable x whose initial value is distributed over the interval [0,1], the y-axis represents the global time. The plot shows that Realyst also computes that x must be at least 0.25 at time 0. In total, 57 flowpipes were constructed during forward analysis, 51 of which overlapped with the specification. The result calculated with our method is exactly as we calculated because the defined flow is linear, therefore during the computation of the first segments during forward flowpipe construction, the convex hull of each initial set and its time successor was not (de)bloated.

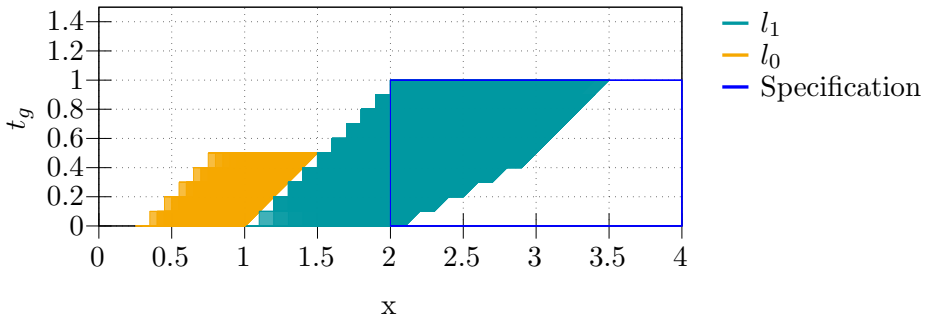


Figure 6.2: Backwards refined segments based on overapproximative forward analysis for automata in Figure 6.1 with time bound 1.

Simple Case B

For the second benchmark, we make one small change to the automaton. As can be seen in Figure 6.3, the reset of the first jump is changed to $x = 5$. Therefore, the post-jump value of x is independent of its pre-jump value.

We define the specification

$$\begin{aligned} x &\geq 5.5 \\ t_g &\leq 1 \end{aligned}$$

and the uniform probability distribution over [0,1] for r_1.

Estimate: Estimating the result here is straightforward and intuitive. There is only one jump between the two locations, which is triggered by the random clock r_1. When the jump is taken, the value of x is set to 5 regardless of its previous value. The time point at which the jump is taken is solely dependent on the sampled value

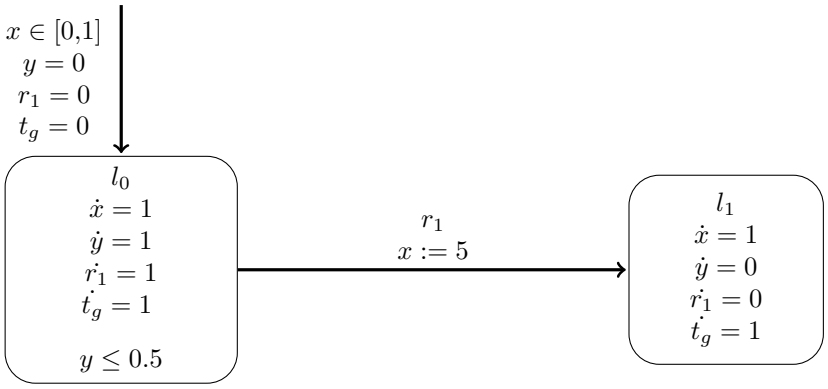


Figure 6.3: Simple SLHA with two locations and a stochastic jump between them. There are three continuous variables x,y and a global timer t_g, and one stochastic variable r_1.

for r_1. We can calculate x_{end}

$$x_{end} = 5 + t_1$$
$$t_1 = T - t_0$$

We know that $r_1 \in [0,1]$. Therefore, the jump can be taken at any point in time, thus $t_1 \in [0,1]$ and the specification is reachable from any x value in the initial set, resulting in a probability of 100% As listed in Table 6.1, we get the same result from the backwards refinement. Figure 6.4 show the flowpipes computed during forward analysis, and the plot in Figure 6.5 shows all backwards refined segments with the x-axis showing x and the y-axis showing the global time. Out of 57 flowpipes computed during forward analysis, 27 intersect with the specification.

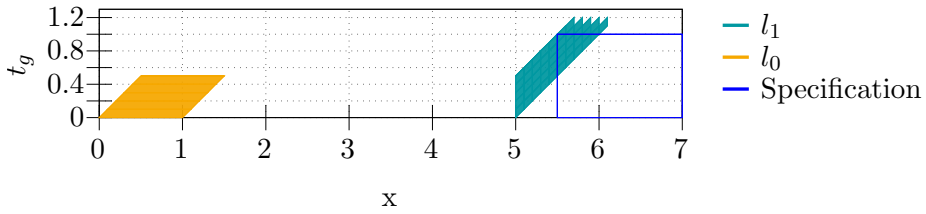


Figure 6.4: Forward flowpipes of a run of automaton in Figure 6.3 defined in simple benchmark B.

Simple Case C

Next, we consider a more complex automata that is still intuitive to estimate. In the automaton, shown in 6.3, there are still two locations connected by a stochastic jump. Two continuous variables, x and y, are defined to be chosen from an interval and both are included in the reset. The reset function of both x and y depends on the previous value of the respective variable.

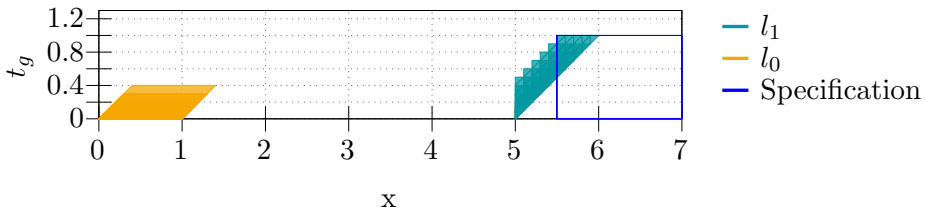


Figure 6.5: Backwards refined segments based on overapproximative forward analysis run of automaton in Figure 6.3 defined in simple benchmark B.

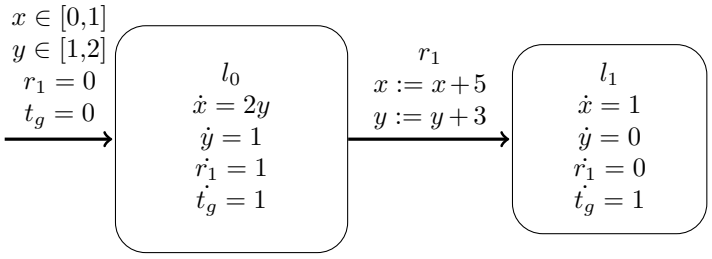


Figure 6.6: Simple SLHA with two location and a stochastic jump between them. There are three continuous variables x,v and a global timer t_g and one stochastic variable r_1.

The specification is defined as:

$$\begin{aligned} x &\geq 15 \\ y &\geq 5 \\ t_g &\leq 2 \end{aligned}$$

We define a uniform probability distribution for the stochastic dimension r_1 over [0,2]. For the continuous dimensions x and y, we define a uniform probability distribution over [0,1] and [1,2], respectively.

Estimation: The value of y is only changed in the flow of l_0 and the reset of the jump.

$$\begin{aligned} y_{end} &= y_{init} + t_0 + 3 \\ 5 &\leq y_{init} + t_0 + 3 \\ t_0 &\geq 2 - y_{init} \end{aligned}$$

Inserting the minimal value 1 for y_{min}:

$$\begin{aligned} 5 &\leq 1 + t_0 + 3 \\ t_0 &\geq 1 \end{aligned}$$

We deduct, that for y, that specification is reachable with $t_0 \geq 1$. The computation for x is more complex, as the flow of x in l_0 depends on y. Since the flow defines the

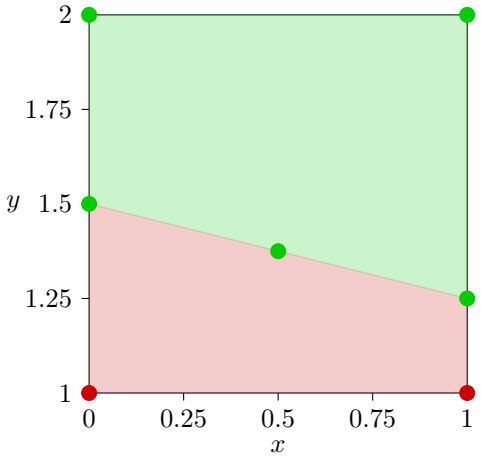


Figure 6.7: Value pairs of x and y from the initial set tested for their ability to reach the specification. From the value pairs, we can deduce that values in the green region can reach the specification, and those in the red region cannot.

derivation of x over time, we can use the integral over the change of x to estimate the final value of x, x_{end}:

$$x_{end} = \int_0^{t_0} 2(y_{init} + t)dt + x_{init} + 5 + (T - t_0) \tag{6.1}$$

Solving the integral and inserting the values from specification results in:

$$2 \cdot y_{init} \cdot t_0 + {t_0}^2 + x_{init} + 5 + (2 - t_0) \geq 15 \tag{6.2}$$

In order to get an estimate of the subset of the initial set, that leads to a goal satisfying state, we find valuations of x and y that still allow for $t_0 \leq 2$.
By inserting values from the initial state set in Equation 6.2, we can estimate the minimum value pairs (x,y) that fulfill the specification.

$$\begin{aligned} x = 0, y = 1 &\Rightarrow t_0 \geq 2.37 \\ x = 1, y = 1 &\Rightarrow t_0 \geq 2.19 \\ x = 0, y = 1.5 &\Rightarrow t_0 \geq 2 \\ x = 0.5, y = 1.375 &\Rightarrow t_0 \geq 2 \\ x = 1, y = 1.25 &\Rightarrow t_0 \geq 2 \\ x = 0, y = 2 &\Rightarrow t_0 \geq 1.7 \\ x = 1, y = 2 &\Rightarrow t_0 \geq 1.5 \end{aligned}$$

From the computed values, we can deduce a subset of the initial set shown in Figure 6.7. The green area covers 62.5% of the initial set. With a uniform probability distribution for all values in the initial set, we calculate an initial-state probability of 62.5%.

Our algorithm uses discrete time steps and computes a probability of 62.47% (62.29%) based on overapproximative (underapproximative) forward analysis, as listed

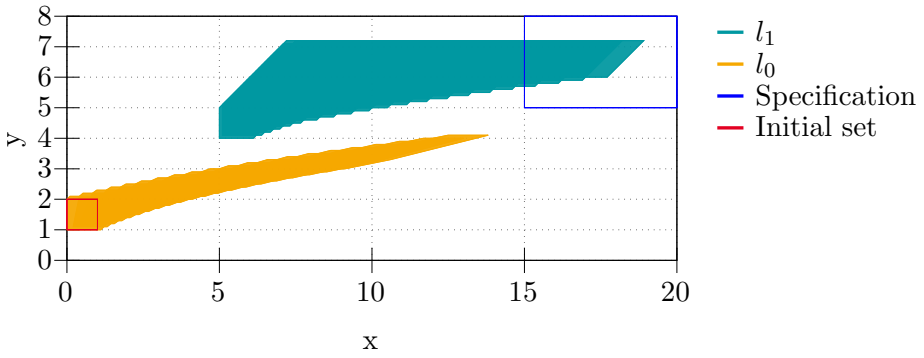


Figure 6.8: Flowpipes of forward analysis for automata in Figure 6.6 with time bound 2.

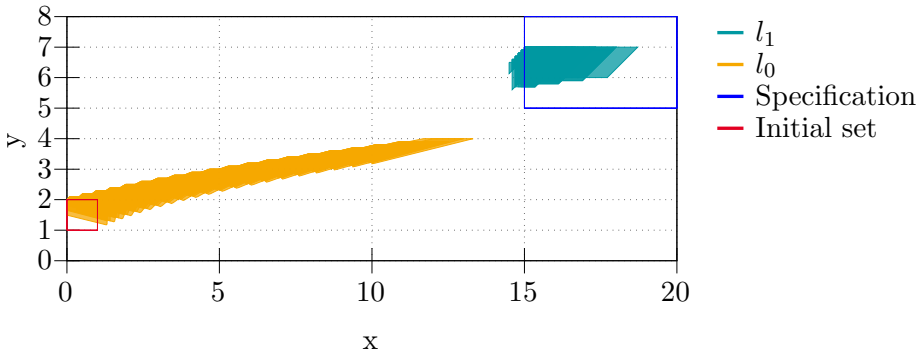


Figure 6.9: Backwards refined segments based on overapproximative forward analysis for automata in Figure 6.6 with time bound 2.

in Table 6.1. Figure 6.8 shows all forward flowpipes of an automaton run, and Figure 6.9 shows the backwards refinement of the goal-satisfying segments. During the forward reachability analysis, 252 flowpipes were constructed. Of these, 21 flowpipes reached the specification and were therefore backwards refined.

Simple Case D

In the next automaton, shown in Figure 6.10, two locations are connected by a stochastic jump. Two continuous variables, x,y, are defined to be chosen from an interval. For both x and y, a reset function is defined in the stochastic jump. The reset function of y depends on the pre-jump value of x, and x is reset to 5. The specification is defined as:

$$\begin{aligned} x &\geq 5 \\ y &\geq 20 \\ t_g &\leq 2 \end{aligned}$$

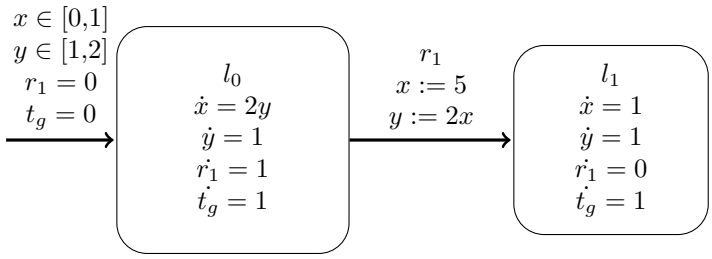


Figure 6.10: Simple SLHA with two location and a stochastic jump between them. There are three continuous variables x,y and a global timer t_g and one stochastic variable r_1.

We define a uniform probability distribution for the stochastic dimension r_1 over [0,2]. For the continuous dimensions x and y, we define a uniform probability distribution over [0,1] and [1,2], respectively.
Estimation: As the value of x is reset to 5 in the reset, the specification value of x is reachable from any initial value. Analogously to Equation 6.1 for Benchmark C, we can estimate y_{end} with

$$y_{end} = 2 \cdot (\int_0^{t_0} 2(y_{init} + t)dt + x_{init}) + (T - t_0) \tag{6.3}$$

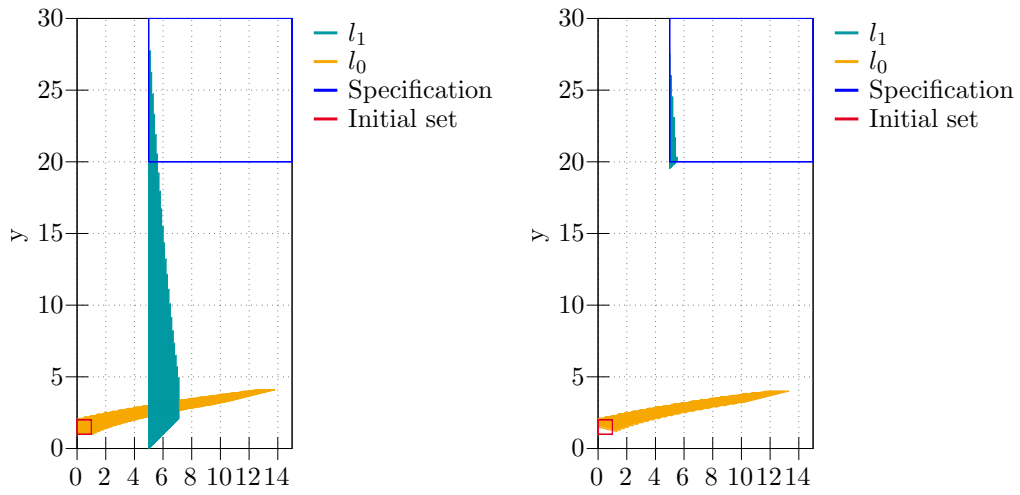
Solving the integral and inserting the values from specification gives the equation:

$$2 \cdot {t_0}^2 + 4 \cdot y_{init} \cdot t_0 - t_0 + 2 \cdot x_{init} + 2 \geq 20 \tag{6.4}$$

As before, we check for valuations from the initial set that satisfy Equation 6.4, thus reducing the valid subset of initial set.

$$\begin{aligned} x_{=}0,\ y = 1 &\Rightarrow t_0 \geq 2.34 \\ x = 1,\ y = 1 &\Rightarrow t_0 \geq 2.17 \\ x = 0,\ y = 1.5 &\Rightarrow t \geq 2 \\ x = 0.5,\ y = 1.375 &\Rightarrow t \geq 2 \\ x = 1,\ y = 1.25 &\Rightarrow t \geq 2 \\ x = 0,\ y = 2 &\Rightarrow t_0 \geq 1.72 \\ x = 1,\ y = 2 &\Rightarrow t_0 \geq 1.58 \end{aligned}$$

With the computed values, the valid subset covers 62.5% of the initial set. With a uniform probability distribution over the initial state set, we get an initial state probability of 62.5%. Our algorithm returns 62.45% (62.21%) for the initial-state probability based on overapproximative (underapproximative) forward flowpipe construction. During the forward reachability analysis, 252 flowpipes were constructed. Of these, 21 flowpipes reached the specification and were therefore backwards refined. The computed forward flowpipes are shown in Figure 6.11a, and the according backward refined segments in Figure 6.11b.



(a) Flowpipes of forward analysis.

(b) Backwards refined segments.

Figure 6.11: Forward Flowpipes and backwards refined goal-satisfying segments for automata n Figure 6.10 for simple Benchmark D.

6.1.2 Complex Benchmarks

The examples in this section are too complex to provide an estimation through computation by hand. We still provide an intuition as to why the computed results are correct. We demonstrated with the simple benchmarks that all used mechanisms function correctly. Therefore, the assumption that the computed results are correct is reasonable.

Case A

The automata in Figure 6.12 has three continuous variables x,y and t_g, one stochastic variable r_1 and five locations. There is one stochastic jump between l_0 and l_2 and three non-stochastic jumps with source and target locations $l_0 \to l_1$, $l_1 \to l_3$, $l_2 \to l_4$, respectively.

We define the specification

$$x \geq 4$$
$$y \geq 8$$
$$t_g \leq 2$$

We explicitly state the time bound of 2. The expiration time of r_1 is sampled from values [0,2].

From the flow specifications of the automaton, we can infer that flowpipes in l_1 and l_4 are most likely to fulfill the specification. However, to get to location l_4 the guard $y \geq 12$ must not be violated. To rise to this value, some time must be spent in an earlier location, potentially not leaving enough time in l_4 for x to increase to 4. In location l_1 it seems likely that the specification will be achieved as x must already

be 4 to jump to l_1 and y increases further while in l_1. However, reaching $x \geq 4$ in l_0 is only likely to succeed with a high initial value of x and y.

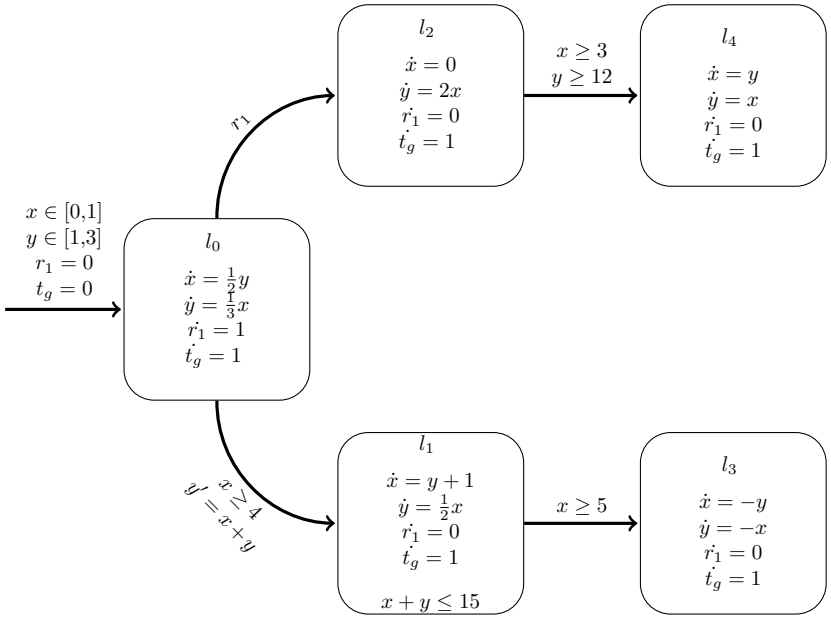


Figure 6.12: SLHA with five location, three continuous and one stochastic variables. One of the continuous variables represents a global timer.

The backward analysis returns the probabilities 8.03% and 7.67% based on the overapproximative and underapproximative forward analysis respectively. Figure 6.13 shows all backwards refined segments based on the overapproximative forward analysis. Of the 286 computed forward flowpipes, 29 intersected with the specification. The low initial-state probability is reasonable. Figure 6.14a shows the forward flowpipes in locations l_0 and l_2. l_2 can be reached from every point in the initial set. However, the jump to location l_4 is never enabled due to $y < 12$. The forward flowpipes of l_2 never intersect with the specification, so none of the flowpipes in l_2 are eligible for backward refinement. Figure 6.14b shows the forward flowpipes in locations l_0, and l_1. We can omit the flowpipes for l_3 because the the values of x and y decrease there. Thus, if the specification has not been reached by the time we jump to l_3, it will not be reached there either. The jump to l_1 is enabled if $x \geq 4$. The number of forward flowpipes in l_0 that fulfill this constraint at least partially is relatively small. Considering the flow of x in l_0 as $\dot{x} = \frac{1}{2}y$, the initial values of x and y must be selected from the higher intervals of their defined initial spectrum to reach $x \geq 4$. This aligns with the result of the backwards refinement, visualized in Figure 6.13.

Complex B

The SLHA in Figure 6.15 has four continuous variables (including the global timer) and two stochastic variables. The initial state for x, y, z is defined over an interval each.

We define, that the stochastic variables r_1 and r_2 to be sampled from [0,2] with a uniform distribution. For the continuous variables, we define uniform distributions

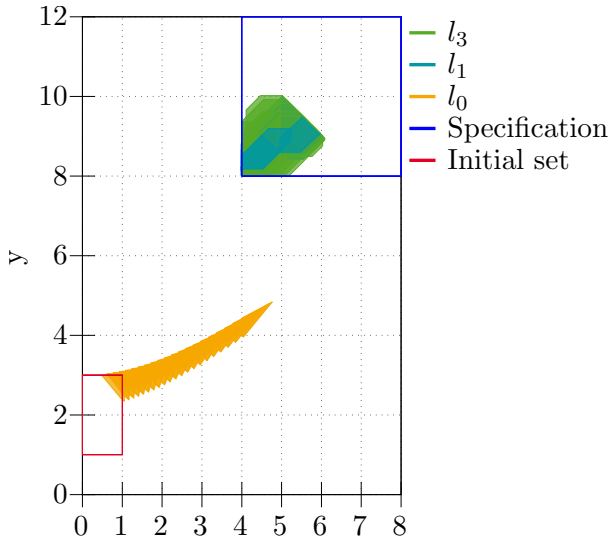
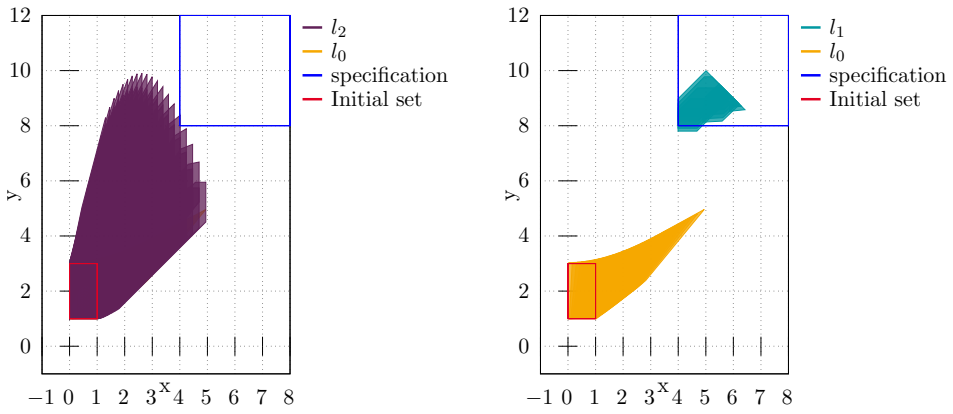


Figure 6.13: Traces of backwards refined segments based on overapproximative forward analysis for automata in Figure 6.12.

over their respective initial sets. The specification is set to:

$$x \geq 4$$
$$z \leq 3$$
$$t_g \leq 2$$

With the time bound of two, 2560 forward flowpipes are constructed during the overapproximative forward reachability analysis, shown in Figure 6.16a. These flowpipes represent the dimensions of x and z. As can be seen, the specification is reachable in a small subset of the flowpipes in location l_3. Applying backwards refinement, computes the segments in Figure 6.16b for dimensions x and z. The segments show, that the specification can be reached via location l_1 but not via l_2. The initial-state probability, computed through backwards refinement, comes to 17.04% (17.01%). The visualization in Figure 6.16b suggests a higher percentage as a bigger subset of the initial set is covered by the backwards refined segments, however, it does not include the variable y. The value of y impacts the flow of x in l_0 and l_1 and the flow of z in l_2. It is also included in the guard of the jump between l_1 and l_2. The crucial role of y in fulfilling the specification lies in l_1. The forward flowpipes in l_2 fail to reach an x value above 4. The flow of l_2 for x is defined by the constant 1. In l_1 x is assigned the y-dependent flow $\dot{x} = y$. Therefore, a bigger initial value of y supports reaching the specification of x. Figure 6.17a shows the backwards refined segments with dimensions x and y. Only initial values for y that are on the higher end of its initial spectrum are returned by the backwards refinement. Figure 6.17a analogously displays the backwards refined segments with dimensions y and z. Taking the dependencies between all three variables into account the initial-state probability of 17% is reasonable.



(a) Forward flowpipes in locations l_0, l_2.

(b) Forward flowpipes in locations l_0, l_1.

Figure 6.14: Forward flowpipes for automata in Figure 6.12 for complex benchmark A.

Benchmark	t_f	t_b	t_i	p_e	p_o	p_u	Automaton
Simple A	0.57	1.77	0.82	75	75	75	6.1
Simple B	0.67	1.21	0.79	100	100	100	6.3
Simple C	3.49	1.89	2.29	62.50	62.47	62.29	6.6
Simple D	3.07	1.84	2.51	62.50	62.48	62.21	6.10
Complex A	203.42	27.16	5.03	-	8.03	7.67	6.12
Complex B	5744.88	930.14	117.09	-	17.04	17.01	6.15

Table 6.1: Benchmark results with t_f as the duration of forward reachability analysis, t_b as the duration of backwards refinement analysis in seconds, t_i as the duration of Monte-Carlo integration. Forward reachability and backwards refinement both include the over- and underapproximative analysis durations. p_e as the manual probability estimation, and p_o and p_u as approximations of the probability in percent based on over- and underapproximative forward flowpipes, respectively.

6.2 Threats to Validity

We showed the conceptual correctness of the algorithm and presented results on various benchmarks. However, some sources of threats to validity remain.

No calculation proof for complex benchmark: The reachability problem for LHA is undecidable. This counts for forwards and backward reachability. While for small benchmarks, approximations of the results by hand were provided, for more complex examples, such a calculation cannot be given. We have to rely on the correctness of the method and on the correctness of the small foundational benchmarks to assume correctness of the more complex benchmarks. We are also not aware of another tool capable of performing backward refinement on LHA. Therefore, no comparison to existing tools can be made.

No industry example: The provided benchmarks are artificial models designed for the purpose of analysis only. None of the provided benchmarks represent an indus-

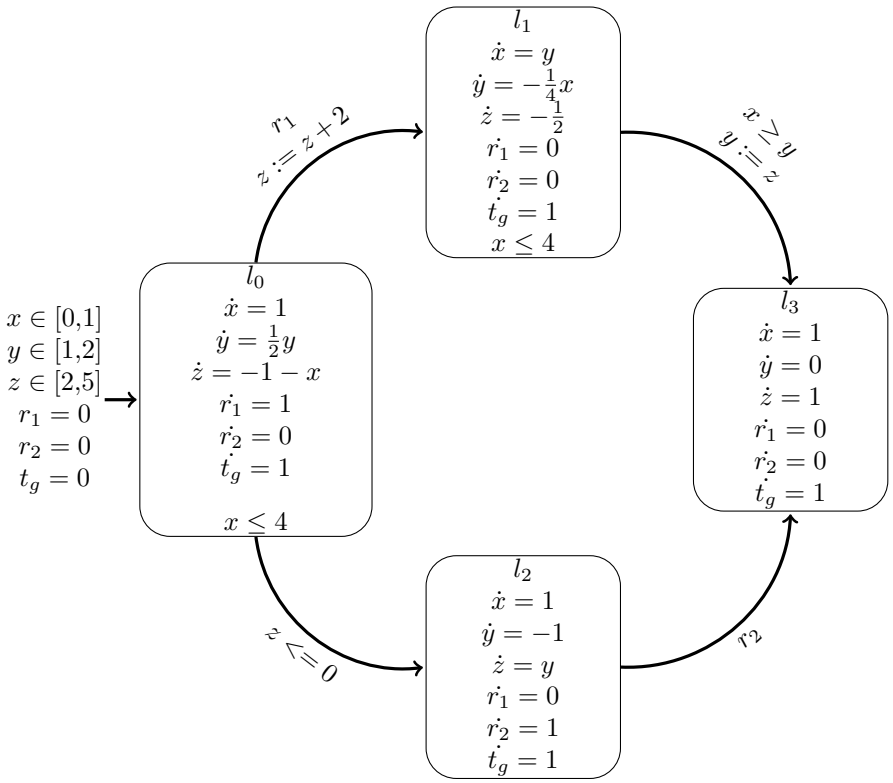
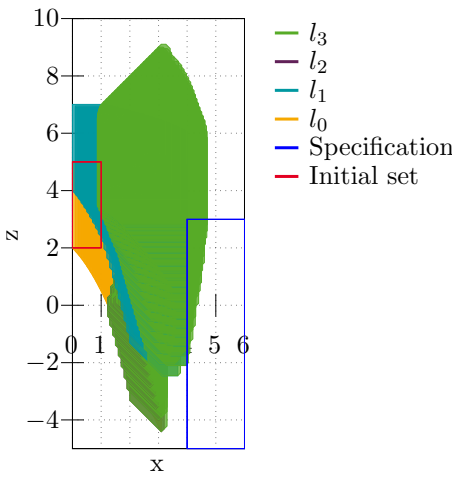
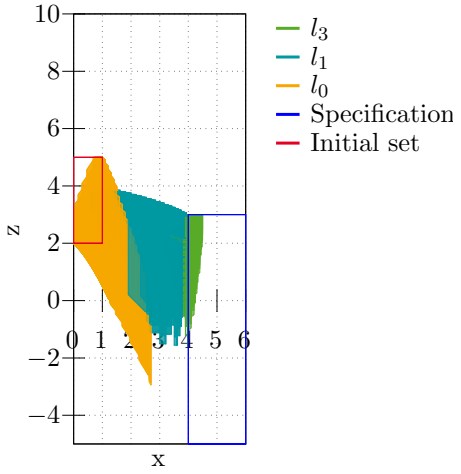


Figure 6.15: SLHA with four locations, four continuous x,y,z and t_g and two stochastic variables r_1,r_2 explored for complex benchmark B.

trial system. We do not have such systems available. Furthermore, the computation time of reachability analysis of HA of industrial systems is still too high to be feasible. Therefore, such an example cannot be provided.

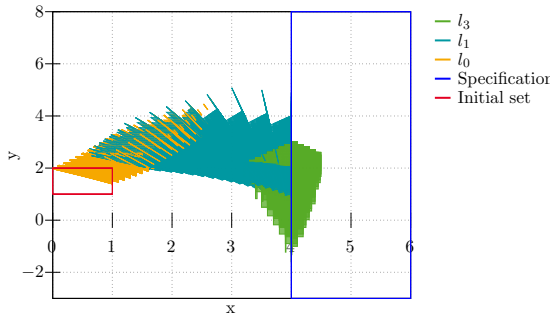


(a) Forward flowpipes with dimension of variables x and z.

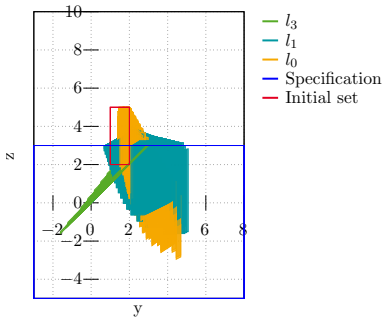


(b) Backwards refinement of goal-satisfying segments with dimension of variables x and z.

Figure 6.16: Forward flowpipes and backwards refinement for complex Benchmark B with Automaton 6.15.



(a) Segment with dimension of variables x and y.



(b) Segments with dimension of variables y and z.

Figure 6.17: Backwards refinement of goal-satisfying segments from a run of Automaton 6.15

Chapter 7

Discussion

In this chapter, we show how the concepts presented in this thesis fit into the current academic context by discussing some related work. We also present possibilities for future work to extend and improve the proposed method.

7.1 Related Work

In this section, we introduce two publications within the current academic context that provide further insight into the topic of reachability analysis in HA.

7.1.1 Maximizing Reachability Probabilities in Rectangular Automata with Random Events [DRÁ+25]

Delicaris et al. introduce RAE, and an algorithm on how to transform them into RAC. The two types of automata were introduced in Chapter 2. In the same publication, they propose a method to maximize reachability probabilities for RAC with history-dependent prophetic scheduling. They define a scheduler which turns RAEs into purely stochastic models which resolves all non-determinism in the automaton. They perform a forward reachability analysis through flowpipe construction for RAC. Part of their preparation for forward analysis, is unrolling the automaton. During backward refinement, one has to keep track of the expiration times of random clocks. Therefore, an expired clock cannot be reset to zero after a jump is taken or its value is lost. This becomes especially relevant when a stochastic jump is taken multiple times as the associated random clock fires multiple times with possibly different expiration times. To overcome this issue, copies of the random clocks are made, such that for every time a stochastic jump (associated with random clock r) is triggered, a different copy of r is used. Thus we keep the expiration of every copy and no value is lost. After the forward analysis, backwards refinement of the goal-satisfying segments is applied. They compute the backwards time closure and jump predecessors of each goal-satisfying valuation set. The backwards refinement returns a fragment of the reach tree that leads to goal-satisfying valuation sets from which a sample domain is extracted. With the sample domains, a polytope can be built, that contains all values of the random clocks that lead to a successful state.

This work solves a similar problem as we do. Delicaris et at. operate on RAC and solve for maximum reachability. For RA – stochastic and non-stochastic – bounded

reachability is solvable while for LHA we can only approximate the reachable states causing a more involved approach to reachability analysis. In our work, we utilize backwards refinement to solve initial state non-determinism in LHA probabilistically. We elaborate on solving other forms of non determinism in Section 7.2.

7.1.2 Scaling Up Reachability Analysis for Rectangular Automata with Random Clocks [SRÁ25]

In this publication, Stuebbe et al. propose three methods to improve the efficiency of time-bounded reachability analysis for RAC by

1) Improving the efficiency for backward computation as it was introduced in [DRÁ+25]

2) Providing the theoretical proof that backwards refinement can be omitted to determine maximum reachability probability for unrolled automata

3) Improving the bounds of numerical integration in an automated fashion.

The second finding is most important to us. With ample bookkeeping about the history of the random variables, backwards refinement can be omitted. While this work only regards RAC, it is a fair assumption that the same would apply to SLHA since the random variables are handled in the same way. However, this conjecture would influence future work on this topic, as we discuss in the next section.

7.2 Future Work

The proposed method solves the problem of initial state non-determinism in SLHA using underapproximative backwards refinement. The applied method ensures correctness building a solid foundation that allows for expansion and optimization. Some opportunities for this are introduced in the following.

7.2.1 Optimize Redundant Computations

The current implementation of the proposed method ensures correctness but some redundancy remains during computation. One such opportunity lies in the segments used initially for backwards refinement. As shown in Chapter 5, we use all segments that overlap with the specification. This ensures that all paths from initial set to the specification are covered, however this results in redundancy. We remarked that it can occur that some backwards refined segments do not intersect with the initial set of the node. This is due to the global time covered by the initial set and the flowpipes. As explained, this does not cause any valuation sets to be lost because all goal-satisfying valuations within the time bound will be included in another segment. However, this causes redundant computations which is not easily fixed. For a goal-satisfying segment, we do not know the global time it covers. With an additional timer as a continuous variable, it is possible to determine the covered time, however, processing this information and cutting the segment down to the relevant part also brings computational effort. Exploring and assessing efficient approaches to remove redundancy exceeds the bounds of this thesis, but it would be necessary if this approach were used in an industrial setting.

7.2.2 Multithreading and Clustering

For forward flowpipe construction multiple configurations of Hypro can improve the runtime of the algorithm. Reachability analysis can be spread across multiple threads to significantly speed up computation. Forward flowpipes can also be clustered to reduce computational effort. For more details, we refer to [Sch19]. Although backwards refinement does not yet implement multithreading, the implementation allows for this extension as each trace can be processed individually. The effects of clustering during forward analysis on backwards refinement have not been explored in this thesis, but it could provide valuable computational improvements.

7.2.3 Enabling Maximum Reachability Probability Computation

The method introduced in this thesis calculates the probability that a state chosen from the initial state set *can* lead to a state in the specification. Whether a chosen state *will* lead to one also depends on other types of non-determinism, such as the expiration time of the random clocks.

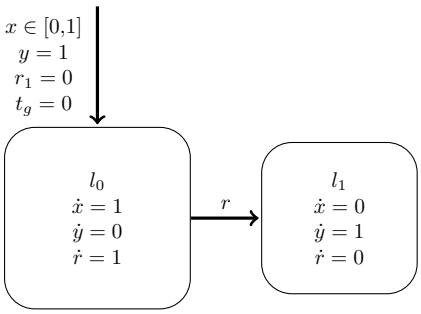


Figure 7.1: Simple SLHA with two location and a stochastic jump between them. There are three continuous variables x,v and a global timer t_g and one stochastic variable r_1.

For example, in the automaton in Figure 7.1, we define the specification

$$x \geq 1.5$$
$$y \geq 1.2$$

We define r to be sampled from the interval [0,2] and fix the time bound to 2.

The initial state probability in this scenario would be 100% because from every state in the initial set, the specification can be reached. However, if the expiration time $s_r = 1$ is sampled, runs with the $x_{initial} < 0.5$ will not be successful, since no state set will intersect with the specification.

In Section 7.1, we discussed [DRÁ+25] to compute maximum reachability prophetically. Overcoming the explained shortcomings of our computed probability would include the utilization of a comparable approach for SLHA. In [SRÁ25], which we also introduce in Section 7.1, Stuebbe et al. determined that for RAC, backwards refinement is only necessary to analyze unrolled automata. With the conjecture, that this also applies to SLHA, continuous non-determinism can be resolved by good bookkeeping of the expiration times during the automata run in unrolled automata. *Realyst* implements a version of this approach. After forward flowpipe construction, the segments that overlap with the specification are extracted. Due to the unrolling of the automaton, the segments store the expiration times of the random clocks that led to the goal-satisfying segment. To enable using automata, that were not unrolled, we would need to extend backwards refinement to the stochastic dimensions. Our approach provides a solid foundation to implement this operation. The proposed backwards reachability method performs backwards refinement on all dimensions. So

far, we only keep the resulting segments that overlap with the initial set of the automaton.

One possibility for extension would be to use this foundation to compute the probability over the stochastic dimensions. In order to do so, we would have to alter the segments that are stored throughout the backwards refinement. The result segments we use at the moment, provide no insight into to the "fire-time" of the random clocks, since in the initial set, they are all set to 0. Instead, the pre-jump segments would be of interest. Since the random variable is reset to 0 in the stochastic jump that was triggered by its expiration, the pre-jump segments hold the information, we are looking for: The time that caused the clock to fire. However, simply switching the segments and dimensions we integrate over, does not suffice. For example, we have to differentiate between fired and non-fired clocks. For instance, it is possible, that a location has two outgoing stochastic jumps triggered by different random clocks. In the pre-jump segment on its own, we do not know which clock fired. Therefore simply using the values of both clocks for the integration domain, may falsify the result for the random clock that did not fire. Using the unrolled approach is an easily utilized method. The missing step, is to connect the initial state probability to the probability of the stochastic dimensions. Whether it suffices to multiply the two probabilities, or some dependability must be taken into account, exceeds the bounds of this thesis.

Chapter 8

Conclusion

We proposed a method to apply backward refinement to LHA in order to solve initial-state non-determinism probabilistically. The method uses the previously computed forward flowpipes. First, we identify the flowpipes that intersect with specified goal states. Then, we use them as the starting points for our analysis. Each segment is backwards refined until the root of the reach tree is reached. We iteratively compute backwards time steps by applying the reverse flow until the initial set of the current node is reached. Then, we perform a backwards jump by applying a reverse reset to reach the parent node in the reach tree. This results in a set of segments that intersect with the initial set of the automaton. Using Monte-Carlo Integration, the computed overlapping segments, and the initial state probability distributions, we calculate the probability that a sampled initial value from the initial state set can lead to a goal-satisfying segment. The probability does not take other sources of non-determinism into account. Even with the sampled value that can reach a goal-satisfying state, there is still the possibility that for some random clock a value is sampled that prevents the automaton run from succeeding.

We successfully demonstrated that the proposed method is computationally correct and implemented our method using the tools Realyst and Hypro. We presented benchmarks of different complexities and demonstrated computational correctness where possible. We presented the computation time of exemplary benchmarks for forwards and backwards analysis as well for Monte-Carlo Integration. For more complex automata with bigger state spaces, more forward flowpipes are typically computed, thus causing a prolonged duration of forward analysis. Similarly, the duration of backwards refinement heavily depends on the number of goal-satisfying segments which is likely to increase with more complex automata as well.

While the presented computations provide good approximations of the initial-state probability, the overall efficiency and scalability require further attention to be considered for industrial applications. As discussed in Chapter 7, there are multiple possibilities to speed up computation. These range from configuration changes, such as using clustering and multithreading, to more complex implementation improvements such as eliminating redundant computations. Furthermore, we looked at some related work that raised the question of how to include further sources of non-determinism in the backwards refinement.

This thesis provided the theoretical foundation for backward refinement for SLHA as well as an implementation of the proposed method. The feasibility of the method was shown on multiple benchmarks of various complexity. We showed that, with backward refinement, the non-determinism introduced through an initial set can be solved probabilistically by providing an initial-state probability. Thus, this work provides the foundation for further academic research in the field of maximum reachability analysis for SLHA.

Appendix A

Commands

The benchmarks provided in Chapter 6 can be run with the commands listed below using Realyst.

Simple Benchmark A:
```
./bin/realyst -t 1 -d 1 -b BACKWARDS_LTI -m A --LTIAnalysis --
-backwardsRefinement=1
```

Simple Benchmark B:
```
./bin/realyst -t 1 -d 1 -b BACKWARDS_LTI -m B --LTIAnalysis --
-backwardsRefinement=1
```

Simple Benchmark C:
```
./bin/realyst -t 2 -d 1 -b BACKWARDS_LTI -m C --LTIAnalysis --
-backwardsRefinement=1
```

Simple Benchmark D:
```
./bin/realyst -t 2 -d 1 -b BACKWARDS_LTI -m D --LTIAnalysis --
-backwardsRefinement=1
```

Complex Benchmark A:
```
./bin/realyst -t 2 -d 2 -b BACKWARDS_LTI -m E --LTIAnalysis --
-backwardsRefinement=1
```

Complex Benchmark B:
```
./bin/realyst -t 2 -d 2 -b BACKWARDS_LTI -m F --LTIAnalysis --
-backwardsRefinement=1
```

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995. Hybrid Systems.

[BBH06] Manuela L. Bujorianu, Henk A.P. Blom, and Holger Hermanns. Functional abstractions of stochastic hybrid systems. In Christos Cassandras, Alessandro Giua, Carla Seatzu, and Janan Zaytoon, editors, *Analysis and Design of Hybrid Systems 2006*, IPV–IFAC Proceedings Volume, pages 160–165. Elsevier, Amsterdam, 2006.

[DK05] Pedro R. D'Argenio and Joost-Pieter Katoen. A theory of stochastic systems part i: Stochastic automata. *Information and Computation*, 203(1):1–38, 2005.

[DRÁ+25] Joanna Delicaris, Anne Remke, Erika Ábrahám, Stefan Schupp, and Jonas Stübbe. Maximizing reachability probabilities in rectangular automata with random events. *Science of Computer Programming*, 240:103213, 2025.

[DSÁR23] Joanna Delicaris, Stefan Schupp, Erika Ábrahám, and Anne Remke. Maximizing reachability probabilities in rectangular automata with random clocks, 2023.

[DSSR24] Joanna Delicaris, Jonas Stübbe, Stefan Schupp, and Anne Remke. *RealySt: A C++ Tool for Optimizing Reachability Probabilities in Stochastic Hybrid Systems*, pages 170–182. 01 2024.

[Egg14] Andreas Eggers. *Direct handling of ordinary differential equations in constraint-solving-based analysis of hybrid systems.* PhD thesis, Universität Oldenburg, 2014.

[Hen00] Thomas A. Henzinger. *The Theory of Hybrid Automata*, pages 265–292. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[Kop96] Peter William Kopke. *The Theory of Rectangular Hybrid Automata.* PhD thesis, Cornell University, 1996.

[Pet78] G Peter Lepage. A new algorithm for adaptive multidimensional integration. *Journal of Computational Physics*, 27(2):192–203, 1978.

[SÁE22] Stefan Schupp, Erika Ábrahám, and Tristan Ebert. Recent developments in theory and tool support for hybrid systems verification with hypro. *Information and Computation*, 289:104945, 2022.

[SÁMK17] Stefan Schupp, Erika Ábrahám, Ibtissem 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, 04 2017.

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

[SRÁ25] Jonas Stübbe, Anne Remke, and Erika Ábrahám. Scaling up reachability analysis for rectangular automata with random clocks, 2025.

[WKA24] Mark Wetzlinger, Adrian Kulmburg, and Matthias Althoff. Inner approximations of reachable sets for nonlinear systems using the minkowski difference. *IEEE Control Systems Letters*, 8:2033–2038, 2024.

[WRÁ23] Lisa Willemsen, Anne Remke, and Erika Ábrahám. Comparing two approaches to include stochasticity in hybrid automata, 2023.

[WRÁ25] Lisa Willemsen, Anne Remke, and Erika Ábrahám. *(de-)Composed And More: Eager and Lazy Specifications (CAMELS) for Stochastic Hybrid Systems*, volume 3 of *Lecture Notes in Computer Science*, pages 309–337. Springer, 2025.

[XSE16] Bai Xue, Zhikun She, and Arvind Easwaran. Under-approximating backward reachable sets by polytopes. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 457–476, Cham, 2016. Springer International Publishing.