Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

BACHELOR OF SCIENCE THESIS

USING THE STAR SET REPRESENTATION FOR THE ANALYSIS OF STOCHASTIC HYBRID AUTOMATA

Sinan David Lindemann

Communicated by
Prof. Dr. Dr.h.c. Erika Ábrahám

Examiners:
Prof. Dr. Dr.h.c. Erika Ábrahám
Dr. -Ing. Andre Stollenwerk

Additional Advisor:
József Kovács

Aachen, 29.12.2025

Abstract

The analysis of *stochastic hybrid automata* is increasingly challenged by the growing complexity of real-world processes, requiring efficient computational methods. *Star sets* have been proposed as a representation capable of improving computational efficiency, potentially enabling the analysis of larger and more complex models. This work investigates whether the use of star sets can enhance the efficiency of reachability analysis within `RealySt`, a tool built on `HyPro`. To this end, `RealySt` was templated and modified to allow the use of star sets. Two stochastic hybrid automata models are analysed, comparing the performance of star sets against two alternative *state set representations*, $\mathcal{V}$*-polytopes* and $\mathcal{H}$*-polytopes*. The results demonstrate that star sets can be applied within `RealySt` for the analysis of stochastic hybrid automata, although current performance is limited due to redundant conversions and the lack of optimisation for many processes. These findings highlight both the feasibility of using star sets in this context and the potential for significant efficiency improvements with further development, offering insights for future research and practical use.

Contents

Chapter 1

Introduction

Modern technical systems, industrial workflows, and automated production environments are growing steadily in complexity. This development is driven not only by increasing global demand, but also by the integration of advanced technologies, interconnected components, and autonomously acting subsystems[Mon14] (as seen, for example, for autonomous driving[AD14][MHR17] or highly automated production lines[RRK10][BBHM05]). As a result, ensuring the safety and reliability of such systems has become a central challenge in both industry and research[SÁW+24]. One established approach to address this challenge is the modelling of system dynamics as hybrid automata, which combine both discrete and continuous behaviour. When uncertainties or probabilistic influences are present, these models are extended to stochastic hybrid automata, allowing for a more realistic representation of real-world processes.

The verification of hybrid[Sch19] and stochastic hybrid automata[HLS00] typically relies on *reachability analysis*[1], which aims to determine the probability with which a system can reach *unsafe* states from a given initial configuration. However, as models grow in dimension and complexity, the computations required for reachability probability analysis become more demanding. In particular, operations such as *Minkowski sums* can become computationally expensive when relying solely on classical representations for state sets, such as $\mathcal{H}$-polytopes and $\mathcal{V}$-polytopes[Tiw07]. This has motivated the search for alternative state set representations that offer better performance for operations required in reachability analysis.

Among these candidates, the star set representation has received increasing attention[AMÁ23][BD17]. Star sets provide highly efficient ways to encode high-dimensional sets while supporting operations that are fundamental to reachability computations. Transformations and intersections with *halfspaces* are among the operations for which star sets are particularly advantageous. If star sets can accelerate the underlying computations, larger and more complex models may become feasible for reachability analysis, thereby increasing the practical relevance of formal verification approaches for real-world systems.

The aim of this work is to investigate whether star sets constitute a suitable and potentially more efficient alternative for the analysis of stochastic hybrid automata. To assess this, the tool `RealySt`, which relies on the `HyPro` library for its state set operations, was extended and templated to support additional set representations,

[1]There are other interesting approaches, such as Monte Carlo based techniques[BEOB14][EP08].

thereby laying the foundation for further developments in the future. As part of this work, star sets were integrated into the toolchain and adapted for use in the existing workflow.

To evaluate the impact of this integration, the performance of star sets is compared with that of $\mathcal{H}$-polytopes and $\mathcal{V}$-polytopes using two benchmark models. The first benchmark is a small-scale example used to validate correctness and basic functionality, ensuring that the new representation integrates consistently with the existing components. As a second benchmark, an electric car model presented in [DSÁR23] is used. It models an electric vehicle with multiple charging modes and stochastic driving times and, due to its larger size, serves as a more realistic benchmark. Although the current implementation is not yet fully optimised, the evaluation provides insights into the usability of star sets in this context. Overall, this work contributes an initial investigation of star sets within `RealySt` for the verification of stochastic hybrid automata. The results highlight both the current limitations arising from implementation overheads and the considerable potential for further optimisation. These findings provide a basis for further improvements to star sets and `RealySt`.

1.1 Related Work

The analysis of hybrid and stochastic hybrid automata has been the subject of extensive research, with various state set representations and tool implementations developed to improve efficiency and robustness. Although none of the existing work employs star sets in `RealySt`, three works are particularly noteworthy.

There are multiple approaches to reachability probability estimation for stochastic hybrid systems. While this work focuses on estimations based on reachability analysis, [DDL$^+$12] proposes a simulation-based approach in which the probability is estimated from samples obtained through repeated representative runs.

The `HyPro`[SÁMK17] library provides a comprehensive framework for reachability analysis, supporting multiple state set representations, including $\mathcal{H}$-polytopes and $\mathcal{V}$-polytopes. `HyPro` forms one of the foundational tools used in this thesis.

More recently, star sets were proposed for reachability analysis and implemented in [Tam21] for `HyPro`. These implementations serve as the basis of the implementations later introduced in this work in regards to star sets.

Star sets have previously been applied to reachability analysis in neural network verification [Mas23], demonstrating their suitability for efficient analysis.

Finally, `RealySt`[DSSR23] is another notable tool that focuses on stochastic hybrid systems, providing an analysis environment capable of handling probabilistic behaviour in hybrid automata.

In contrast, this thesis introduces and evaluates the use of star sets within `RealySt`, building on previous work while exploring their applicability in this context.

1.2 Outline

In chapter 2, the most important theoretical foundations of hybrid automata and reachability analysis are introduced. Furthermore, the different relevant state set representations for this thesis are briefly explained in that chapter. After a general introduction to the theoretical aspects of the topic, chapter 3 provides an overview of the library `HyPro`, with a focus on the most important methods in relation to its adaptation

for compatibility with `RealySt` and star sets. Next, chapter 4 introduces the tool `RealySt` and outlines its general workflow. Subsequently, the new implementations are presented in chapter 5, along with current challenges in the implementation and some ideas for future projects. This is followed by a detailed presentation of two benchmarks used to test the current implementation in chapter 6. Finally, the results of this work are summarised, discussed, and various possible future improvements are outlined in chapter 7.

Chapter 2

Preliminaries

2.1 Hybrid Automata

A *hybrid system* is one that combines the behavioural characteristics of both *discrete* and *continuous* systems. In discrete systems, changes occur instantaneously, while in continuous systems, variable dynamics are described by *ordinary differential equations* governing their evolution over time. Hybrid systems are capable of modelling a wide range of processes found in the natural world. One important application is their use in industrial settings, such as the modelling of steel plants [Feh99], where ensuring correct functionality is of considerable importance [NDN+16].

To formally model and analyse such systems, *hybrid automata* are commonly employed. For clarity, hybrid automata are first introduced in their general form, followed by an important subclass and its stochastic extension.

2.1.1 General Hybrid Automata

The hybrid automaton consists of various interconnected parts. The set of locations Loc represents the discrete control modes of the system, corresponding to different operational states, such as heating or waiting. The set of variables Var describes properties of the underlying system, whose values evolve continuously over time and may also be subject to discrete updates, such as resets. The continuous evolution of a variable $v \in Var$ is described by its first derivative, denoted as $\dot{v}$, while the value of v after a discrete change is denoted by v'. The labels Lab allow the automaton to distinguish and synchronise discrete transitions by associating them with observable events, while internal transitions may be represented by a special stutter label τ. In each location, a predicate is assigned to define the continuous evolution ($Flow$), by relating the variables to their derivatives, thereby inducing the corresponding system of ordinary differential equations. This flow may continue as long as the invariant of the location remains valid. The invariant of a location is given by a predicate $\varphi \in Inv$, which constrains the permitted values in that location. If the invariant is violated, the automaton cannot remain in the current state and must undergo a discrete transition. A jump $(l,a,g,r,l') \in Edge$ consists of a source location $l \in Loc$, a synchronization label $a \in Lab$, a guard $g \in Pred_{Var}$, a reset $r \in Pred_{Var,Var'}$, and a target location $l' \in Loc$. The jump can only be taken if the guard is active (i.e. the predicate is fulfilled) and if the valuation of the variables after applying the reset satisfies the

invariant of l'. If the jump is taken, the automaton continues in the target location. $Init$ defines the initial states $\sigma_i = (Loc \times \mathbb{R}^d)$ of the automaton as a set of pairs consisting of valid start locations and corresponding valuations of all variables. These parts together as a tuple define the hybrid automata (Formal definition in 2.1.1).

Definition 2.1.1 Hybrid Automata [Sch19]

The *hybrid automaton*, is defined by a tuple:

$$\mathcal{H} = (Loc\ ,Var\ ,Lab\ ,Flow\ ,Inv\ ,Edge\ ,Init)$$

- Loc - a finite set of locations.
- Var - a vector of variables $Var = (x_0, \ldots, x_{d-1})$,
  d being the dimension of $\mathcal{H}$. $\dot{Var}$ denotes the set of first derivatives of the continuous change, while Var' represents the set of values after discrete changes.
- Lab - a finite set of labels, including the stutter label τ.
- $Flow : Loc \to Pred_{Var \cup \dot{Var}}$ [a] - specifies the continuous evolution of the variable values within a location according to differential equations.
- $Inv : Loc \to Pred_{Var}$ - specifies the Invariant for each location.
- $Edge \subseteq Loc \times Lab \times Pred_{Var} \times Pred_{Var,Var'} \times Loc$ - defines the transitions or jumps $((l,a,g,r,l') \in Edge)$, with $l \in Loc$ as its source location, $a \in Lab$ as the synchronization label, $g \in Pred_{Var}$ as the jumps guard, $r \in Pred_{Var \cup Var'}$ as the reset, and $l' \in Loc$ as the target location.
- $Init : Loc \to Pred_{Var}$ - defines the initial states.

[a] $Pred_X$ denotes the set of all predicates over the set X.

With the formal definition of the automaton established, the behaviour of the automaton also needs to be specified. For this, the operational semantics is introduced in 2.1.2, which defines how states evolve through continuous flow and discrete jumps.

Definition 2.1.2 Operational Semantics[Sch19]

One step of the hybrid automata $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$ is defined by the following operational semantics:

- The rule for the flow:

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

- The rule for the jump:

$$\frac{\begin{array}{c} e = (l,a,g,r,l') \in Edge \quad l,l' \in Loc \quad v,v' \in \mathbb{R}^d \\ v \models g \quad v,v' \models r \quad v' \models Inv(l') \end{array}}{(l,v) \xrightarrow{e} (l',v')}$$

Based on the operational semantics, which defines how individual states evolve over time, it is now possible to describe complete system behaviours. For this the definition of a *path* from [Sch19] is introduced in 2.1.3.

Definition 2.1.3 Path[Sch19]

A *path* π of $\mathcal{H}$ is a sequence of states of $\mathcal{H}$ that are connected through alternating flow and jump steps:

$$\pi = \sigma_0 \xrightarrow{\tau_0} \sigma_1 \xrightarrow{e_1} \sigma_2 \xrightarrow{\tau_2} \sigma_3 \xrightarrow{e_3} \dots$$

with $\sigma_i = (l_i,v_i) \in Loc \times \mathbb{R}^d$ being states of $\mathcal{H}$, $\tau_i \in \mathbb{R} \geq 0$,$e_i \in Edge$, and $v_0 \models Inv(l_0)$. A path is *initial* if additionally $v_0 \models Init(l_0)$.

To illustrate the concept of hybrid automata, an explanatory hybrid automaton is shown in Fig. 2.1 and formally described as a tuple in 2.1.4. The hybrid automaton models a water cistern that can be automatically filled over time and manually emptied through human interaction. The initial state is labelled filling, and the variable w, representing the water level, can take any value between 5 and 7 litres. In the filling state, the water level gradually increases as time, denoted by t, passes. The system can remain in this state only while $w \leq 7$. Once the value $w = 2$ is reached, the guard for the transition to the flushing state, representing the release of water, is activated and the transition may be taken. If this transition is taken, the water level decreases until it reaches 0. Due to the state invariant and the existing transitions, the system must then return to the filling state. If the cistern fills completely, the system may instead take the transition to the waiting state, where the water level remains constant. From the waiting state, a transition to the flushing state may be taken at any time.

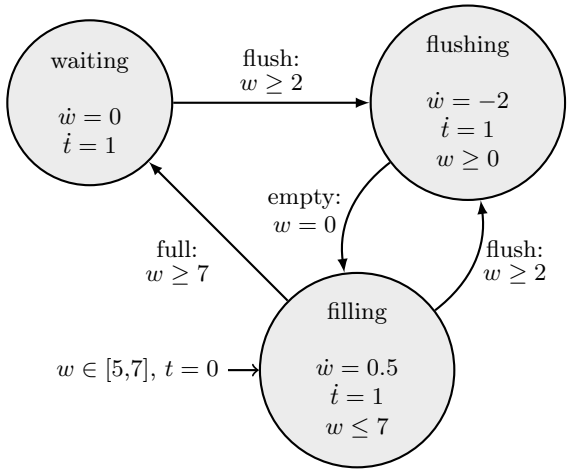


Figure 2.1: The example illustrates a typical water cistern, such as one commonly found in most households.

Example 2.1.4 Formal Definition

- $Loc = \{\text{waiting, flushing, filling}\}$
- $Var = \{w, t\}$
- $Lab = \{\tau, \text{full, flush, empty}\}$
- $Flow(\text{waiting}) = \{(f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall s \in \mathbb{R}_{\geq 0}. f(s)(w) = c),$
  $(f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall s \in \mathbb{R}_{\geq 0}. f(s)(t) = s + c))\}$
- $Flow(\text{flushing}) = \{(f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall s \in \mathbb{R}_{\geq 0}. f(s)(w) = -2s + c),$
  $(f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall s \in \mathbb{R}_{\geq 0}. f(s)(t) = s + c))\}$
- $Flow(\text{filling}) = \{(f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall s \in \mathbb{R}_{\geq 0}. f(s)(w) = 0{,}5s + c),$
  $(f : \mathbb{R}_{\geq 0} \to V | \exists c \in \mathbb{R}. \forall s \in \mathbb{R}_{\geq 0}. f(s)(t) = s + c))\}$
- $Inv(\text{flushing}) = \{v \in V | v(w) \geq 0\}$
- $Inv(\text{filling}) = \{v \in V | v(w) \leq 7\}$
- $Edge = \{$
  $(\text{waiting, flush,}\{(v, v') \in V^2 | v(w) \geq 2 \wedge v(w) = v'(w) \wedge v(t) = v'(t)\}\text{,flushing}),$
  $(\text{flushing, empty,}\{(v, v') \in V^2 | v(w) = 0 \wedge v(w) = v'(w) \wedge v(t) = v'(t)\}\text{,filling}),$
  $(\text{filling, flush,}\{(v, v') \in V^2 | v(w) \geq 2 \wedge v(w) = v'(w) \wedge v(t) = v'(t)\}\text{,flushing}),$
  $(\text{filling, full,}\{(v, v') \in V^2 | v(w) \geq 7 \wedge v(w) = v'(w) \wedge v(t) = v'(t)\}\text{,waiting})\}$
- $Init = \{(\text{filling, v}) \in \Sigma | v(w) \in [5,7], v(t) = 0\}$

2.1.2 Rectangular Automata

While the previous section introduced hybrid automata in their general form, many analysis and verification techniques rely on restricted subclasses that ensure decidability, as discussed in[HKPV95]. One such subclass is that of rectangular automata[KJ96]. The rectangular automaton is a hybrid automaton, "in which not only invariants and guards, but also all assertion predicates, encode *rectangular sets*"[Sch19]. The rectangular set is defined as a combination of constraints comparing a variable to constants. The flow is now defined with a rectangular predicate in the form of $\dot{x} = [a,b]$ with $x \in Var$ and $a,b \in \mathbb{N}$. Similarly, the predicate for the jumps is now restricted to $x' = [a,b]$ and even guards or invariants can be written in the rectangular form $x \in [a,b]$. In the following the rectangular predicates of a set X will be written as $Pred^R_X$. Definition 2.1.5 shows the formal definition of the rectangular automata, with the changes highlighted.

Definition 2.1.5 Rectangular Automata[Sch19][KJ96]

The *rectangular automaton*, modelling a hybrid system, is defined by a tuple:

$$\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$$

- Loc - a finite set of locations.
- Var - a vector of variables $Var = (x_0, \ldots, x_{d-1})$,
  d being the dimension of $\mathcal{H}$. $\dot{Var}$ denotes the set of first derivatives of the continuous change, while Var' represents the set of values after discrete changes.
- Lab - a finite set of labels, including the stutter label τ.
- $Flow : Loc \rightarrow Pred^R_{Var \cup \dot{Var}}$ - specifies the continuous rectangular evolution of the variable values within a location by constraining the derivatives to a rectangular set.
- $Inv : Loc \rightarrow Pred^R_{Var}$ - specifies the Invariant for each location in a rectangular form.
- $Edge \subseteq Loc \times Lab \times Pred^R_{Var} \times Pred^R_{Var \cup Var'} \times Loc$ - defines the transitions or jumps $((l,a,g,r,l') \in Edge)$, with $l \in Loc$ as its source location, $a \in Lab$ as the synchronization label, $g \in Pred^R_{Var}$ as the jumps guard in a rectangular form, $r \in Pred^R_{Var'}$ as the rectangular reset, and $l' \in Loc$ as the target location.
- $Init : Loc \rightarrow Pred^R_{Var}$ - defines the initial states with the corresponding values as a rectangular set.

Predicates such as $x = c$, $x \doteq c$, $x \leq c$, $x \geq c$ can remain unchanged, as they already define rectangular sets by comparing a variable to a constant. Alternatively, they could be rewritten to conform more closely to the expected form. For example, $x = c$ can be transformed into $x \in [c,c]$. In the following, this transformation will not

be applied, as it does not improve readability.

2.1.3 Stochastic Hybrid Automata

Although classical hybrid systems provide a powerful modelling framework for complex systems that combine discrete logic with continuous dynamics, many real-world applications are also affected by uncertainty and random influences, such as fluctuating environments or measurement noise. To capture these aspects, *stochastic hybrid systems*[HLS00] extend hybrid systems by incorporating stochastic behaviour into their components, thereby enabling more realistic modelling for real-world problems. As all benchmarks in chapter 6 involve rectangular automata, this chapter considers only their stochastic extensions. For further details on stochastic singular automata, refer to [DSÁR23], and for general stochastic hybrid automata, see [HLS00].

Definition 2.1.6 Stochastic Rectangular Automata[DSÁR23]

The rectangular automaton extended with *random clocks* is defined by its tuple:

$$\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init, Var_R, Flow_R, Edge_R, Distr_R)$$

$(Loc, Var, Lab, Flow, Inv, Edge, Init)$ is unchanged as in 2.1.5.

- Var_R - a finite set $Var_R = \{r_1, \ldots, r_d\}$ of random clocks.
- $Flow_R : Loc \to \{0,1\}_{Var_R}$ - specifies the flow of the random clocks, indicating for each location whether a clock is active (1) or inactive (0).
- $Edge_R \subseteq Loc \times Var_R \times Loc$ - defines the stochastic jumps $e = (l,r,l')$, where l is the source location and l' being the target location. For each set (l,l') a different random clock r is used.
- $Distr_R : Var_R \to \mathbb{F}_c$ - assigns a continuous probability distribution to each random clock r, defining a sequence $S_{r,0}, \ldots, S_{r,n}$ of values over time that r must reach in order for the corresponding jump to be taken.

The stochastic rectangular automata uses the added clocks to account for *time nondeterminism*[1]. This is achieved by associating each stochastic jump with a random event governed by a continuous probability distribution $\mathbb{F}_c$ and a random clock. While a jump is active, the corresponding stochastic clock is activated and advances with a constant rate of 1[2]. The distribution is used to generate an expiration date for the random clock to reach, at which point the jump is taken. Once the random clock reaches expiration time, the jump occurs and the clock is reset to 0. The new random value to be reached is generated from the distribution as the next expiration time[DSÁR23]. This forms a sequence $S_{r,0}, \ldots, S_{r,n}$ for each random clock r, whose values are independent samples drawn from the underlying probability distribution.

[1] Definiton 2.1.7

[2] And its derivative is 0 if the jump is not enabled.

The rectangular automaton with random clocks 2.1.6 allows for four different types of nondeterminism defined in 2.1.7.

Definition 2.1.7 Types Of Nondeterminism[DSÁR23]

- *Initial nondeterminism* arises from the choice among multiple possible initial states.
- *Time nondeterminism* occurs because time can progress in a location while discrete transitions are enabled.
- *Rate nondeterminism* occurs because the continuous variables in a location can change according to a range of possible rates, as allowed by the flow conditions.
- *Discrete nondeterminism* arises when multiple jumps are enabled at the same time.

The example in Fig. 2.1 was modified to include two random clocks, in order to capture stochastic flushing behaviour Fig. 2.2. The first random clock r represents a flush that may occur when the cistern is being filled, while the second clock c models a flush with a filled cistern. Each clock is only enabled in the locations where the corresponding flushing behaviour is possible. The clock c is governed by a stochastic distribution of $Distr_c \sim Uniform[0,100]$, while the clock r follows an exponential distribution $Distr_r \sim Exp(0.005)$. This reflects the fact that a flush occurring while the cistern is being filled is highly unlikely, but may occur at any point over a long time horizon, whereas a flush with a filled cistern is guaranteed to occur within finite time, provided that the waiting state is eventually reached.

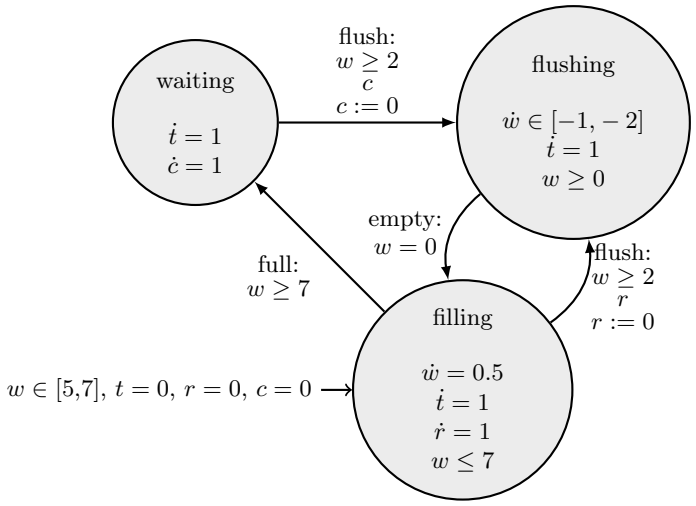


Figure 2.2: The Cistern automaton modified with random clocks r and c. Derivatives equal to 0 are omitted for readability.

Stochastic hybrid automata allow the modelling of uncertainties, which is especially beneficial when estimating the probability of events occurring in complex systems, such as failures or delays, in a factory setting. By combining continuous dynamics with probabilistic discrete transitions, these models capture both physical processes and random events, such as external influences. This makes them a powerful tool for analysing reliability and risk in real-world systems[LP10].

2.2 Reachability

The hybrid automaton shown in Fig. 2.1 can now be analysed to make predictions about the states the automaton may occupy at a given point in time. In general, the goal is to determine whether an undesirable state or a desired state is *reachable*, i.e. whether there exists any initial path that can lead the system into the given bad/good state.

This process is known as *reachability analysis*. In simpler cases, such as the example shown above, it is possible to determine all the potential states in which the system may be at a given point in time. In other cases, the automaton may exhibit non-linear behaviour, making predictions through simple calculations unfeasible[HKPV95]. In such cases, the system's behaviour may be abstracted through *over-* or *underapproximation*[Che15][Sch19].

These approximation techniques involve fitting the systems behaviour to more easily calculable geometric forms.[1] When underapproximation is employed, some reachable states may be omitted due to the nature of the underlying computations. Consequently, any state deemed reachable in the abstraction is also reachable in the original system, whereas some states classified as non-reachable in the abstraction may, in fact, be reachable in the original system. With overapproximation, the opposite is true: a reached undesirable state in the abstraction may not be reachable in the original system. However, if no undesirable states are reachable in the abstraction, the system can be considered *safe*.[2]

Similarly, the automaton in Fig. 2.2 can be analysed to estimate the probability of reaching a specific state. One approach to estimating this probability is discussed in chapter 4.

[1] *State Sets Representations* | more in section 2.4

[2] A system is considered safe if the intersection of the states reachable from the initial states and the set of undesirable states is empty.

2.3 Flowpipe Construction

An over-approximative forward approach is the *flowpipe construction based reachability algorithm*[Sch19].

Only a high-level overview of the approach is provided here, while [SNÁ17, LG09] present a more detailed explanation. For better understanding, pseudocode of a reachability algorithm using this approach is provided in 2.3.1.

Algorithm 2.3.1 Forward Reachability Analysis[Kan24]

Input: Initial set $Init$
Output: Set R of reachable states

---

1 $R := Init$
2 $R^{new} := Init$
3 **while** $R^{new} \neq$ **do**
4 Let $stateSet \in R^{new}$
5 $R^{new} := R^{new} \backslash \{stateSet\}$
6 $R' := computeFlowPipe(stateSet)$
7 **if** !$jumpDepthReached()$ then
8 $R^{new} := R^{new} \cup computeJumpSucc(R')$
9 end **if**
10 $R := R \cup R^{new}$
11 end **while**
12 **return** R

The *timestep* size (δ) is used to divide the analysis of the automaton into equally sized time intervals $[0,\delta],[\delta,2\delta],\ldots$. These timesteps are then used to compute the individual overapproximated segments of the flowpipe. By combining these segments, the whole behaviour of the automaton gets approximated.

The first segment is created by transforming $Init$ using the given flow for one timestep. This transformation can be performed by applying the *matrix exponential* $e^{\delta A}$ to X_0[1]. The matrix exponential represents the time evolution of a linear system $\dot{x} = Ax$ where δ is the time interval. Afterwards the computed set $e^{\delta A}X_0$ is bloated by taking its Minkowski sum with a box B to compensate for non-linear behaviour[LG09]. The union (or convex hull[2]) of this set and the initial set then forms the first segment of the flowpipe. This process is visualised in Fig. 2.3.

Definition 2.3.2 Convex Set[Zie95]

A set of points $K \subseteq \mathbb{R}^d$ is convex if with any two points $x,y \in K$ it also contains the straight line segment $[x,y] = \{\lambda x + (1-\lambda)y : 0 \leq \lambda \leq 1\}$ between them.

Further segments are computed via a linear transformation of the prior segment. Due to the over-approximation introduced in the first segment, all subsequent segments remain over-approximative[Tse20].

[1] In the following X_0 denotes the initial states $Init$.
[2] The convex hull of a set is the smallest convex 2.3.2 set that contains all points from the initial set.

When the guard of an edge or transition is satisfied, all sets that fulfil the guard are transformed according to the jump's reset, producing a new initial set from which a new flowpipe is constructed. If the invariant is no longer satisfied, or if the flowpipe intersects with an undesirable state, the analysis terminates.

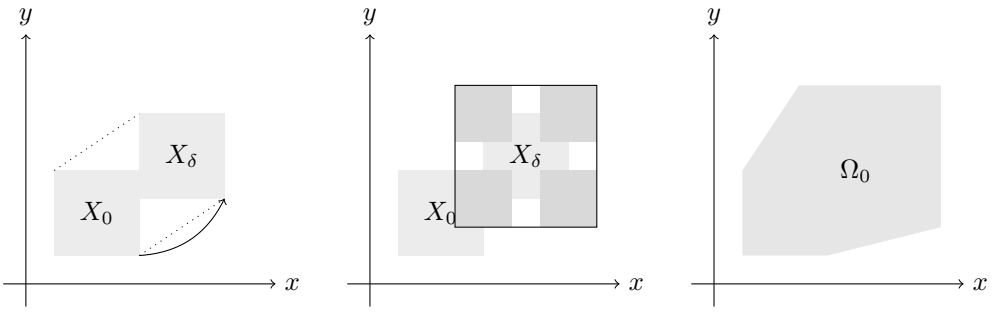


Figure 2.3: Computation of the first segment. In Fig. 2.3a, the initial set X_0 is illustrated together with its transformation under the matrix exponential. The dotted lines represent the naive approach of constructing the first segment, by forming the convex hull of X_0 and X_δ, whereas the dark arrow shows possible non linear behaviour that the naive method would not capture. In Fig. 2.3b X_δ is bloated with a box to include the non linear behaviour, forming the first segment Ω_0 with the convex hull of the union with X_0 in Fig. 2.3c

2.4 State Set Representation

Each state of a hybrid automaton can be described by a geometric representation, either through approximation or, in some cases, directly. These representations must be convex in order to ensure the feasibility of various mathematical operations. The choice of which representation to use depends on a range of factors. Each representation offers different advantages and disadvantages for specific operations, and in some cases must be transformed before it can be used effectively[Sch19]. In general, by using more complex state set representations, it is possible to achieve a lower approximation error at the cost of increased computational effort[Sch19].

2.4.1 Polytopes

Using the definition 2.3.2, it is now possible to define geometric objects based on convexity. A convex polytope [Zie95] is a convex set that can be represented either by a finite set of linear constraints, or by its extreme points [Sch19]. Accordingly, the two important representations of polytopes considered in this work are $\mathcal{V}$*-polytopes* (*vertex-polytopes*) and $\mathcal{H}$*-polytopes* (*halfspace-polytopes*).

Definition 2.4.1 Polytopes [Sch19]

A $\mathcal{V}$-polytope is given by a finite set $P_V = \{v_0, \ldots, v_{m-1}\}$ of vertices $v_i \in \mathbb{R}^d$ of the convex hull of the polytope. A polytope and its set of points S is *convex* if $\forall x,y \in S. \forall \lambda \in [0,1] \subseteq \mathbb{R}. (\lambda x + (1-\lambda)y) \in S$. P_V is defined as

$$P_V = \left\{ x \middle| x = \sum_{i=0}^{m-1} \lambda_i \cdot v_i \wedge \sum_{i=0}^{m-1} \lambda_i = 1 \wedge \lambda_i \in [0,1] \right\}.$$

A d-dimensional convex $\mathcal{H}$-polytope P_H is given by a pair (N,c) with $N \in \mathbb{R}^{m \times d}$ and $c \in \mathbb{R}^m$. This pair defines a convex set

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

where h_i is a d-dimensional halfspace [a] with the i-th row of N as its normalvector and c_i as its offset.

[a] A d-dimensional *halfspace* h is defined as $h = \{x \in \mathbb{R}^d | n^T \cdot x \leq c\}$ for $n \in \mathbb{R}$ (*normalvector*) and $c \in \mathbb{R}$ (*offset*)

An explanatory $\mathcal{V}$-polytopes can be seen in Fig. 2.4a, defined by its vertices:

$$PV = \{(1,0.5),(2,0.5),(1.5,2),(0.5,1.5),(0.3,0.7)\}$$

The $\mathcal{H}$-polytope shown in Fig. 2.4b represents the same polytope as before, expressed equivalently by the following halfspaces:

$$N = \begin{pmatrix} 0 & -0.5 \\ 0.4743 & 0.1561 \\ -0.2236 & 0.4472 \\ -0.4848 & 0.1212 \\ -0.1374 & -0.4808 \end{pmatrix} \text{ with } c = \begin{pmatrix} -0.25 \\ 1.0272 \\ 0.559 \\ -0.0606 \\ -0.3776 \end{pmatrix}$$

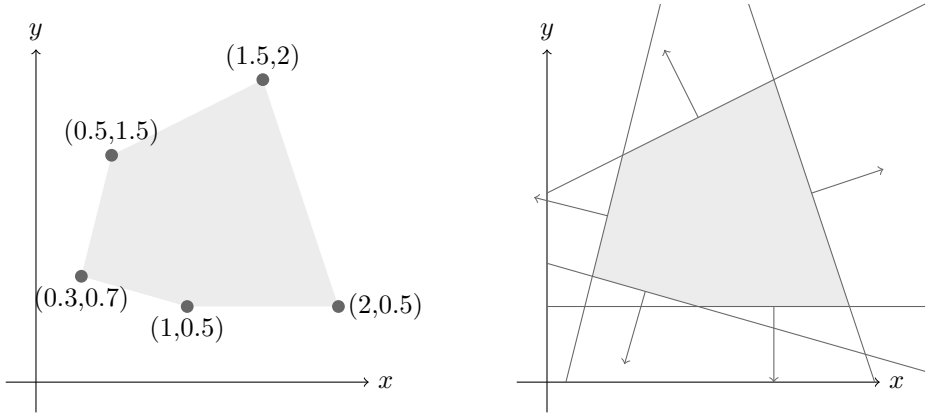


Figure 2.4: The same polytope represented either as a $\mathcal{V}$-polytope Fig. 2.4a or as an $\mathcal{H}$-polytope Fig. 2.4b

.

While both representations describe the same geometric object, it becomes apparent that they have different strengths and weaknesses. The calculation of the *union* is much simpler with $\mathcal{V}$-polytopes, as it is possible to extend the convex hull and remove the unnecessary vertices. In contrast, for $\mathcal{H}$-polytopes, the *intersection* can be computed in polynomial time, as it reduces to solving a small system of linear equations. In some cases, it can be beneficial to convert a polytope into the other representation in order to perform a more efficient operation[1]. The efficiency of these operations is also affected by redundant information. While $\mathcal{V}$-polytopes can contain unnecessary vertices, $\mathcal{H}$-polytopes can contain redundant half-spaces, both types of redundancy may arise during various operations[Fre05]. When these redundancies are removed, the polytopes are referred to as *reduced*.

Converting an $\mathcal{V}$-polytope into its $\mathcal{H}$-representation is called *facet enumeration*. In up to two dimensions, a variety of algorithms are available, such as `Graham's scan` [Gra72]. For higher-dimensional polytopes, `Quickhull` [BDH96] is commonly used. Although these algorithms are quite efficient, converting from $\mathcal{V}$-polytopes to $\mathcal{H}$-polytopes still represents a considerable computational effort.
The reverse direction is known as *vertex enumeration* and can be carried out by solving linear equations. For a d-dimensional $\mathcal{H}$-polytope all intersection points of d equations (halfspace boundaries[2]) are computed and then verified by checking whether they satisfy the remaining halfspace constraints. This also requires considerable computational effort and can negatively impact the efficiency of the analysis.

2.4.2 Star Set

Generalised star sets[DV16] are a more recently proposed and introduced state set representation, which "turned out to be exceptionally good candidates"[AMÁ23], due to their efficient handling of various operations[3].

[1]Example: The union of $\mathcal{H}$-polytopes is usually calculated by first converting to $\mathcal{V}$-polytopes.
[2]Analogous to definition of halfspaces in 2.4.1, using $n^T \cdot x = c$ instead of $n^T \cdot x \leq c$
[3]Proofs of the properties that make star sets so efficient can be found in [AMÁ23]

Definition 2.4.2 Generalised Star set[BD17]

A *generalised star set* Θ is a tuple $\langle c,V,P\rangle$ where $c \in \mathbb{R}^n$ is the centre,$V \in \mathbb{R}^{n\times m}$ is the generator matrix consisting of m vectors $V = \{v_1,\ldots,v_m\}$ where v_i in $\mathbb{R}^n$, called the basis vectors, and $P : \mathbb{R}^m \to \{\top,\bot\}$ is a predicate. The basis vectors are arranged to form the star's $n \times m$ basis matrix. The set of states represented by the star is given as:

$$[\![\Theta]\!] = \{x | x = c + \sum_{i=1}^{m} \alpha_i v_i \text{ such that } P(\alpha_1,\ldots,\alpha_m) = \top\}$$

This current definition allows for non convex star sets such as the one formed by the following tuple $\langle c,V,P\rangle$:

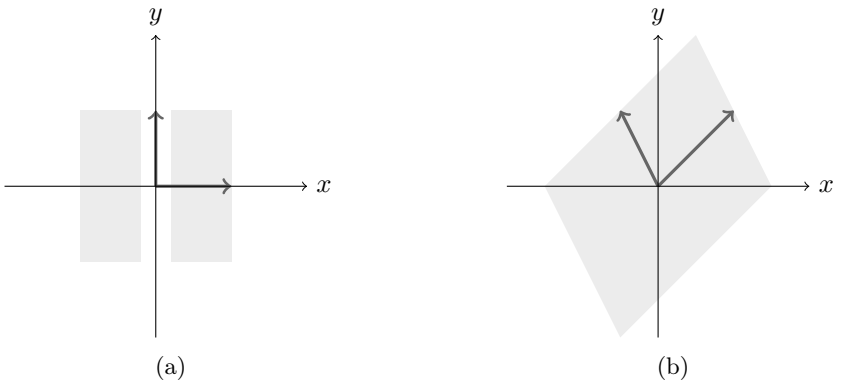
$$c = \begin{pmatrix}0\\0\end{pmatrix}, V = \begin{pmatrix}1 & 0\\0 & 1\end{pmatrix} \text{ and } P(\alpha_1,\alpha_2) \text{ defined as}$$
$$(\alpha_1 \in [-1,-0.2] \wedge \alpha_2 \in [-1,1]) \vee (\alpha_1 \in [0.2,1] \wedge \alpha_2 \in [-1,1])$$

This star set is non convex due to its gap in the middle. Star sets such as these are undesired in the context of this work. Using the restrictions from [AMÁ23] the predicate P is formed:

$$P = \{\alpha \in \mathbb{R}^m | C\alpha \leq d\} \text{ with } d \in \mathbb{R}^p \text{ and } C \in \mathbb{R}^p \times m \text{ for } p \in \mathbb{N}$$

With this restriction P forms a convex polyhedron. Fig. 2.5b shows an example of a convex star set, defined by its centre, generator matrix, and predicate:

$$c = \begin{pmatrix}0\\0\end{pmatrix}, V = \begin{pmatrix}1 & -0.5\\1 & 1\end{pmatrix}, \text{ and } P = \{\alpha \in \mathbb{R}^2 | C\alpha \leq d\}$$
$$\text{with } C = \begin{pmatrix}-1 & 0\\1 & 0\\0 & -1\\0 & 1\end{pmatrix} \text{ and } d = \begin{pmatrix}1\\1\\1\\1\end{pmatrix}$$



(a) (b)

Figure 2.5: A non-convex star set(Fig. 2.5a) and a convex star set (Fig. 2.5b).

Star sets are more efficient than other representations when it comes to *linear transformations*[1]. Intersections with half-spaces can also be computed easily by adding the rewritten constraint to the predicate P, as introduced in [TMLM$^+$19]. The conversion between star sets and other state set representations is primarily realised using the method proposed in [Tam21], which allows for transformations between star sets and $\mathcal{H}$-polytopes[2]. Conversions involving any other representation are performed by first converting the star set to an $\mathcal{H}$-polytope.

The strategy behind converting a star set into an $\mathcal{H}$-polytope is to interpret the star set as the result of a sequence of affine transformations. The generator matrix describes the linear part of these transformations, while the centre represents the offsets. An $\mathcal{H}$-polytope is first created from the predicate and is then transformed using the information provided by the centre and generator matrix. The resulting $\mathcal{H}$-polytope represents the same space as the original star set.

To derive a star set from an $\mathcal{H}$-polytope, the centre c is assigned to be the origin, represented by a zero vector of the appropriate dimension. The matrix V is set to the standard basis of that dimension, and the predicate P is formed from the constraints defining the polytope.

[1] In star sets, *affine transformations* are used, which can be described as a linear transformation followed by a translation[Tam21].

[2] Further details are provided in chapter 3

Chapter 3

HyPro

HyPro[hyp], first introduced in [SÁMK17], is a C++ programming library, designed to support reachability analysis based on flowpipe construction. The central idea of HyPro is to provide a unified and extensible library for representing and manipulating geometric state sets that arise during the reachability analysis of hybrid automata. Instead of focusing on a single state set representation, HyPro was conceived as a flexible platform that accommodates a broad variety of geometric abstractions, each with its own algorithmic strengths and weaknesses, as discussed in section 2.4.
At the time of its initial publication, HyPro already offered eight different state set representations, including $\mathcal{H}$-polytopes and $\mathcal{V}$-polytopes. Each representation in HyPro implements the full set of operations required for reachability computations, including transformations, intersections, unions, and more. The state set representations are implemented on top of a collection of geometric and algebraic data structures provided directly by the library, such as points, vectors, half spaces, and formal hybrid automata.

An important and central feature of HyPro is its support for conversions between different state set representations. Since no single representation is universally optimal with respect to both efficiency and precision, switching between representations during reachability analysis can be advantageous. To support such flexibility, HyPro provides exact and over-approximating conversion routines for most included state set representations. These conversions are implemented through a converter mechanism that is fully templated, allowing users to replace the default conversion strategies with specialised methods if required.

To support an easier empirical evaluation of the implemented state sets, HyPro also includes a reachability algorithm, alongside benchmark examples. These enable users to test and compare different state set representations in isolation, without relying on external methods. However, [SÁMK17] emphasises that users are encouraged to replace generic methods with specialised algorithms that behave more efficiently in the given cases.

All together, HyPro can be characterised as a versatile foundational library for geometric representations and operations in reachability analysis. Its modular design and broad support for different state set representations laid the foundation for later extensions as described in [SÁE22], where a new state set representation was introduced. The star set was added in [Tam21], together with a multitude of functionalities, although not all conversion methods were implemented at that point[1].

[1]This will be discussed later in section 3.1 and chapter 5

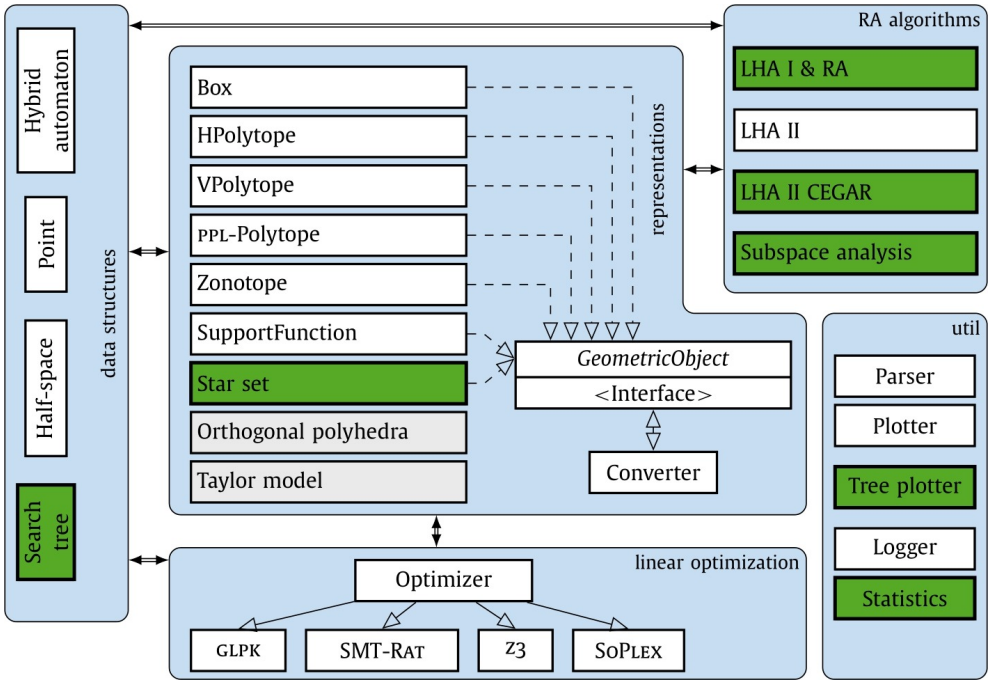


Figure 3.1: The structure of `HyPro` from [SÁE22], with the newest additions at that time highlighted in green.

3.1 Important Implementations In HyPro

Within the context of this thesis, the most significant functionalities are the star set implementation and the associated conversion functions. Accordingly, the majority of modifications introduced to HyPro during this project were focused on these areas[1].

3.1.1 Implementation Of $\mathcal{H}$-Polytopes

To provide a clearer understanding of the underlying concepts, the implementation of the state set representation $\mathcal{H}$-polytope will be discussed first as a representative case. The implementation of the class `HPolytopeT` is one of the earliest and most coherent implemented state set representations in `HyPro`. Over time, most extensions of representations followed a similar design strategy. Thus, `HPolytopeT` is a good illustration of the programming principles used throughout the project. Its structure closely follows the theoretical description provided in section 2.4.1, where an $\mathcal{H}$-polytope is defined as an intersection of halfspaces. This notion is directly mirrored in the implementation, which relies on a collection of halfspace objects to represent the geometric structure.

All data structures in `HyPro`, including $\mathcal{H}$-polytopes, employ extensive templating. Templates are used not only for specifying the numerical type, but also for selecting the appropriate converter and additional configuration parameters[2].

[1] A few minor changes were also made for $\mathcal{V}$-polytopes and points. More on that in chapter 5

[2] Only the number template played an essential role in the later implementation.

The class `HPolytopeT` offers a wide variety of constructors in order to simplify transitions between representation formats, most notably number types. When invoked, these constructors create a new $\mathcal{H}$-polytope with the specified numerical type, converting all numbers accordingly.

A further notable feature of the implementation is the integration of internal caching mechanisms. These caches are designed to accelerate operations that are frequently executed during the reachability analysis. For example, the cache of extremal points (`mVertices`) stores the points obtained through the `Quickhull` algorithm. Since the computation of vertices is comparatively expensive, caching them can yield substantial performance gains for repeated queries, such as unions, and can also make subsequent conversions computationally cheaper.

Although caching leads to a decreased computational cost in most cases, the effect of caching can be diminished if conversions between different state set representations occur frequently. Most conversions invalidate the caches, effectively requiring parts of the computation to be performed from scratch. Keeping the caches for a new representation can be harmful, since approximations may be used.

3.1.2 Implementation Of Star sets

Star sets are implemented in a similar manner to $\mathcal{H}$-polytopes. Their design closely follows the general theoretical description, with the restriction of the predicate to a polytope. This polytope is represented through a set of linear constraints.

The current implementation of star sets encompasses all essential geometric operations. However, many of these operations internally rely on a conversion to an $\mathcal{H}$-polytope representation. While this approach uses the efficiency of the already introduced $\mathcal{H}$-polytope operations, it also introduces substantial computational overhead. To mitigate this potential inefficiency, star sets also employ an internal caching mechanism that stores an alternative representation as an $\mathcal{H}$-polytope. This cache allows operations that depend on the $\mathcal{H}$-polytope conversion to be executed without redundant conversions, increasing the performance for repeated calls.

Several methods, including evaluation and intersection checks, rely heavily on efficient linear programming procedures. The computational cost of these procedures increases with the number of constraints defining the star set, which can become a limiting factor for larger models. To address this challenge, *reduction techniques*[1] can be applied to remove redundant constraints, thereby reducing the computational effort of the linear programming procedures. In the current implementation, only a single method[2] uses such reductions. More frequent use of reductions may lead to improved performance.

The primary conversions implemented for star sets correspond to those described in 2.4.2. These conversions enable the computation of an equivalent $\mathcal{H}$-polytope representation from a given star set, as well as the reverse transformation back to a star set. Other types of conversions in the current implementation are performed indirectly, via an intermediate $\mathcal{H}$-polytope representation. This approach introduces unnecessary overhead, as each conversion involves additional processing and potential caching operations. In frequent or repeated transformations between representations, this becomes a great source of inefficiency.

[1] A common method involves reducing the predicate of a star set to simplify its representation, as demonstrated in [Sch19].

[2] The method `unite`

It is therefore important to acknowledge that, although the current design is functionally complete, it does not optimise a wide range of operations. As a result, the efficiency of reachability analysis with star sets is reduced compared to what might be expected based on the theoretical description of star sets.

Chapter 4

RealySt

The reachability analysis and state sets provided by `HyPro` are used in `RealySt`, introduced in [DSSR23]. `RealySt`[1] is a tool for stochastic reachability which also serves as the main component of this work. It is an open source, `C++` based tool designed for the computation of optimal, time bounded reachability probabilities for subclasses of stochastic hybrid automata. The subclasses of hybrid automata for which `RealySt` was developed are *singular automata with random clocks* and *rectangular automata with random clocks*. These automata are equipped with stochastic timing behaviour. They combine continuous evolution and discrete transitions with stochastic delays through random clocks. Nondeterminism may arise in initial states, flow rates, or timing behaviour in both subclasses. Stochastic transitions correspond to the expiration of random clocks. A more detailed explanation of these subclasses of stochastic hybrid automata can be found in [PSR21] and [DSÁR23].

The tool relies on flowpipe construction based reachability using convex sets and integrates over these sets using the `Monte Carlo VEGAS` algorithm implemented in `GSL`[GDT+02]. In its original version, `RealySt` only supported $\mathcal{H}$-polytopes as its geometric backbone. This work extends its capabilities to additional state set representations by restructuring the tool to be template based.

4.1 Workflow Of RealySt

The operational workflow of `RealySt` can be summarised as a six stage pipeline:

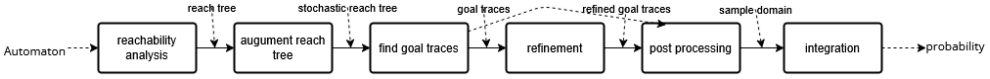


Figure 4.1: An illustration of the program flow of `RealySt` for a given automaton, adapted from [DSSR23].

The operational workflow of `RealySt` follows a structured sequence of transformations that connect reachability analysis with stochastic integration, as illustrated in Fig. 4.1. Starting from a stochastic hybrid automaton, `RealySt` first performs a time

[1]Available at [rea]

bounded reachability analysis in which operations provided by `HyPro` are employed for flowpipe construction, computing all reachable sets. This process generates a *reach tree*[1]. In this process, the stochastic random clocks are treated as continuous behaviour. In the next step, the reach tree is enriched with stochastic information by annotating each node with the status of the random clocks along its path. Based on this stochastically augmented reach tree, `RealySt` identifies all nodes whose associated state sets intersect the goal region[2]. The paths from the root of the tree to the intersection of the goal region and the fulfilling node are called *goal traces* and describe a symbolic execution path that may lead to a successful reachability event. To eliminate errors, `RealySt` proceeds with a *backward refinement* step. Here, the intersections are used to determine which regions can indeed reach the goal. The outcome of this phase is a refined set of goal traces. After the backward refinement has precisely identified the relevant parts of the state space and their trajectories that lead to the goal, the workflow shifts to the probabilistic analysis required to compute the actual reachability probability. The traces describe which behaviours may reach the goal, while the final two steps compute how likely these behaviours are under the given random delays. The first step prepares the integration by deriving a suitable integration domain. Each refined trace imposes constraints on the evolution of the random delays, and from these constraints, the corresponding region for the delays is constructed. All regions are combined into a single integration domain that describes the delay combinations that allow the system to reach the goal. In the final step, `RealySt` performs the integration over this domain to determine the total probability of reaching the goal. This is achieved by applying a *Monte Carlo* simulation[Wei00]. Monte Carlo integration estimates probabilities by randomly sampling points from the underlying distribution and computing the fraction of samples that satisfy the event of interest. Since a finite number of samples is used to estimate the probability, the result is subject to sampling variability, which decreases as the number of samples increases. This deviation from the exact result is called *statistical error*.

4.2 Implementation Of RealySt

Before introducing the extensions developed in this thesis, it is helpful to outline the central implementation aspects of `RealySt` that are essential for understanding the subsequent modifications. In its current form, `RealySt` is organized into several stages, forming the overall workflow. Over time, certain parts of this workflow have already been refactored to increase flexibility with respect to the state set representations employed in the reachability analysis. As a consequence, a preliminary templated architecture was introduced, enabling `RealySt` to support multiple state set representations for performing the reachability computations. The next chapter will delve deeper into the existing templating, exploring its limitations in more detail.

Since this thesis focuses primarily on extending the existing templating mechanisms related to probabilistic integration, the most relevant components of the existing implementation are located in the probability integration modules. These files contain the routines responsible for estimating the probabilistic reachability results, and they form the basis for the modifications developed in this work.

[1] Reach trees are outside the scope of the present work. For a more comprehensive understanding, the exemplary reach trees illustrated in [Hua21] may be consulted.

[2] The goal region defines the set of states that the system aims to reach.

A step preceding the probabilistic integration is the generation of the stochastic reach tree, as mentioned previously. The computation of the nodes of the stochastic reach tree is particularly important for this thesis. In the current implementation, these nodes also lack compatibility with other state set representations. A more detailed view of the implementation is available at [rea]. For the purposes of this work, a general understanding of `RealySt` is sufficient without examining the code.

Chapter 5

Implementation

The implementation work presented in this chapter constitutes the main contribution of this thesis. It addresses several structural limitations in the original architectures of `RealySt` and `HyPro`, and extends both to support additional state set representations for reachability analysis with `RealySt` in stochastic hybrid automata. The following sections first describe the modifications applied to `RealySt`, followed by the extensions introduced in HyPro that were required to enable full compatibility.

5.1 Changes In RealySt

Although `RealySt` was partially extended with a templated architecture intended to make the analysis independent of any particular state set representation, the practical applicability of this design was limited by the absence of templating in the probabilistic integration routines. These routines[1] were written with the implicit assumption that all state sets are represented by $\mathcal{H}$-polytopes.

This design became problematic when attempting to generalise `RealySt` to alternative representations such as the star sets. In particular, one component of the integration pipeline required the state set representation to use the numerical type `double`. A straightforward replacement of the existing representation templates with a `double` based version was infeasible. While integration requires `double`, the reachability analysis itself depends on exact computations performed with rational numbers using the `mpq` class[Gra96]. As a consequence, a unified template parameter for both phases could not fulfil the differing numerical requirements.

To resolve this incompatibility, a third representation template was introduced. The resulting templates are distinguished by their three roles. `RepresentationHypro` is the template for the reachability operations in `HyPro`. `Representation` is the main template for `RealySt`, and `SRepresentation` is the second template for the integration part, which requires `double` as numerical type.

This separation makes it possible to use different numerical types in different stages of the workflow. To ensure consistency between `Representation` and `SRepresentation`, an additional assertion was added to `common.h`, where the state set types are configured. This check ensures, at compile time, that `Representation`

[1] Located in `src/libRealySt/include/librealysr/probabilities/integration.tpp`(and `integration.h`)

and `SRepresentation` refer to the same state set representation, differing only in their underlying number type.

A second major limitation became apparent during the construction of the stochastic reach tree. During runtime, `RealySt` always created stochastic reach tree nodes using $\mathcal{H}$-polytopes, regardless of which representation was actually selected for the reachability analysis. This behaviour was rooted in the original design of the stochastic reach tree node, which hard-coded the expectation that the underlying state set is always an $\mathcal{H}$-polytope. Consequently, if a differing state set representation was used for the reachability analysis in `HyPro`, `RealySt` had to convert the representations from the reach tree nodes to the new representation for the stochastic reach tree nodes. However, the original implementation of this conversion mechanism only supported the case, in which the target representation was an $\mathcal{H}$-polytope, due to the previously mentioned assumption that the representation used for the integration is an $\mathcal{H}$-polytope. To solve this issue, the conversion procedure was abstracted into a `lambda` function. This design allows the stochastic reach tree construction to dynamically adapt to the state set representation actually used in the integration process. Currently, the `lambda` supports $\mathcal{H}$-polytope, $\mathcal{V}$-polytope and star sets. Unsupported representations trigger a dedicated error message, allowing users to immediately identify missing conversion routines for the stochastic reach tree nodes and to extend the system accordingly. This redesign ensures that the stochastic reach tree with its nodes remains compatible with the integration process of `RealySt`.

5.2 Changes In HyPro

The initial phase of work on `RealySt` focused on refining the $\mathcal{V}$-polytope and star set representations to ensure that all necessary methods were present and behaving as expected, both in terms of parameters and returned values. In the case of star sets, this involved the implementation of the *contains* method, which determines if a given point lies in a star set. For most other state set representations, it was also possible to determine if a vector lies in the set. To maintain compatibility with the existing codebase and avoid unnecessary rework, a second contains method was introduced for star sets, overloading the original. As a result, the `contains` function may now be called with either a vector or a point.

There were several additional minor inconsistencies, such as the `removeRedundancy` function, which was implemented as a void method for $\mathcal{V}$-polytopes, whereas all other state set representations returned the modified representation. To maintain consistency across the different state set representations, the function was adapted so that it now behaves in the same manner as its counterparts. Moreover, both star sets and $\mathcal{V}$-polytopes lacked an implementation of the `projectOutConservative` function. This method can be used as an alternative to the standard `projectOut` method to ensure that no reduction of the segment dimension is performed. At present, the conservative version remains incomplete for both star sets and $\mathcal{V}$-polytopes. As a temporary measure, the existing `projectOut` method is called instead. This workaround may lead to less accurate results than anticipated, and further development of the `projectOutConservative` functionality is therefore recommended for future work.

`RealySt` also relies on serialisation through the `cereal` library, which means that every state set representation used within `RealySt` must support serialisation. To

meet this requirement, both star sets and $\mathcal{V}$-polytopes were extended with appropriate serialisation routines that capture all essential information. In the case of $\mathcal{V}$-polytopes, this included the need to serialise the vertices themselves. As the underlying `Point` class did not previously support serialisation, this functionality was added as well, ensuring that all components of the representation can be correctly serialised.

As mentioned in section 3.1, the state set representations require a range of different constructors to allow transformations between parameters, such as the numerical types. The $\mathcal{V}$-polytope implementation lacked a constructor for converting between number types. This transformation is carried out by converting each vertex of the $\mathcal{V}$-polytope into the target numerical type.
A similar issue existed for star sets, which were also missing a suitable number conversion constructor. In this case, the conversion process is somewhat more involved, as it requires transforming the centre, the generators, and the constraints. Converting the centre and the generators is relatively straightforward. However, the constraints require more careful handling. Since the constraints of a star set can be represented by an $\mathcal{H}$-polytope, the conversion is achieved by applying the $\mathcal{H}$-polytope's own number conversion constructor. The resulting star set then possesses the correct numerical type throughout.

Once the modifications had been completed, it was possible to run the simple example[1] using `RealySt`. However, the analysis was returning empty state sets, leading to a segmentation error, whenever the `RepresentationHypro` and the other two representations did not align. Debugging revealed that, among various potential issues, the primary problem stemmed from several methods that lacked proper implementations. Instead of raising an error when invoked, these methods simply returned empty state set representations, making the underlying issue difficult to detect.

The converter from a star set to a *carlpolytope*[2] was corrected by first converting the star set into an $\mathcal{H}$-polytope and then transforming it into a carlpolytope. Likewise, the reverse conversion, from a carlpolytope to a star set, was missing, along with several other conversions involving star sets. Only the conversions required for this thesis were fully implemented, using an intermediate transformation step to an $\mathcal{H}$-polytope. The remaining, unused conversion methods were modified to throw explicit errors, informing the user that the function must be implemented before use. This ensures that future development in this area can proceed more efficiently and with clearer diagnostics.

With these modifications in place, `RealySt` is able to operate correctly using all three state set representations considered in this thesis. In the following chapter, a comparative analysis of these representations will be presented.

[1] Introduced in section 6.1.

[2] The internal details of this representation are not relevant for this thesis. It is only required for the analysis, and therefore the corresponding conversion method must exist.

Chapter 6

Benchmarks

To determine the practical viability of using star sets for stochastic reachability analysis in `RealySt`, this chapter compares them with $\mathcal{H}$-polytopes and $\mathcal{V}$-polytopes using two benchmarks. Throughout the chapter, H denotes $\mathcal{H}$-polytopes, V denotes $\mathcal{V}$-polytopes, and S denotes star sets. Each benchmark is run nine times, testing all possible combinations of the three state set representations. These combinations are denoted in forms such as `HHH` or `HSH`. Here, the first and last letter indicate the Representation and the SRepresentation, while the middle letter refers to the representation used for `HyPro`.

The benchmarks were conducted on a machine equipped with an AMD Ryzen 7 7700X processor with 8 cores and 32 GB of RAM. The experiments were executed inside a virtual machine running Ubunutu 25.04, configured with 4 allocated cores and 22 GB of RAM. The virtual environment was provided by Oracle VirtualBox version 7.2.2, running on a windows 11 host system.

6.1 Simple Running Example

The first benchmark examined is the simple running example provided by `RealySt`[1]. This running example consists of five locations with three variables, two of which are continuous, while the third is a stochastic clock. Owing to its relatively small size, the example serves as an accessible test case for validating the correct functioning of the different state set representations.

The first combination `HHH`, as seen in Fig. 6.1, serving as the control run, resulted in a probability of 0.842701368572847 with an approximate statistical error of $7.306e^{-6}$. All other successful runs for this benchmark produced a similar or same probability and similar statistical error. The entire computation required ≈ 0.046 seconds, with the majority of the time being consumed by the analysis phase. Most computational steps were executed too quickly to be captured by the timers, causing the majority of measurements to register as zero. This significantly complicates any attempt to compare the individual steps in regards to their performance. For a more comprehensive comparison, the second benchmark in section 6.2 was supposed to provide further insight.

[1] [DSÁR23]

```
Probability: 0.842701369572847
Statistical error: 7.306200682064129e-06
Infinity error: 0
[info] 0 Computation Time Complete: 0.0469973 s
[info] Counters and timers:
Counters:
Traces: 2

Timers:
0 Computation Time Complete: 0.0469973 s
1 Model creation: 0 s
2 Analysis: 0.0469973 s
2.1 Reachability Analysis: 0 s
2.2 Augment Reach Tree: 0 s
2.3 Filter Segments: 0 s
2.4 Initialize Trace Structure: 0 s
2.5 Refactor Forward Flowpipe (remove s=0 constraints): 0 s
2.6 Backwards Refinement: 0 s
2.6.2 Compute Intermediate Goal Segment for Backwards Time-
Closure for all Segments: 0 s
2.6.3 Compute Backwards Time Closure for all Segments: 0 s
2.6.4 Perform Backwards Jump for all Segments: 0 s
2.7 Collect segments: 0 s
3 Integration: 0 s
PROBABILITY_INTEGRATION_DIRECT: 0 s
PROBABILITY_INTEGRATION_DIRECT_REMOVE_EMPTY: 0 s
```

Figure 6.1: The baseline run with $\mathcal{H}$-polytopes as the state set representation for both `HyPro` and `RealySt`.

To validate the initial run, `HHH` was executed multiple times afterwards. While the results varied in terms of performance, the outputs remained consistent. The highest observed computation time for `HHH` was approximately 1.5 seconds, whereas the lowest was registered as 0 seconds. Since any comparison would heavily be influenced by which particular run was selected, the comparison will instead use the average and median runtime across ten successful runs. For `HHH`, this resulted in an average computation time of ≈ 0.35 seconds and a median of ≈ 0.24 seconds. The difference between the median and the average for `HHH` indicates that the average is influenced by outliers with longer computation time. This has been done for every combination and the results in Table 6.1 are illustrated in Fig. 6.2, showing the average and median total computation time. Each combination of state set representations had the same or a similar probability and statistical error.

Total Computation Time	HyPro HPolytope	HyPro VPolytope	HyPro Starset
RealySt HPolytope	Average: $\approx 0.35s$ Median: $\approx 0.23s$	Average: $\approx 0.60s$ Median: $\approx 0.30s$	Average: $\approx 0.32s$ Median: $\approx 0.35s$
RealySt VPolytope	Average: $\approx 20.96s$ Median: $\approx 18.73s$	Average: $\approx 14.07s$ Median: $\approx 14.99s$	Average: $\approx 15.77s$ Median: $\approx 15.04s$
RealySt Starset	Average: $\approx 46.90s$ Median: $\approx 48.29s$	Average: $\approx 55.74s$ Median: $\approx 63.55s$	Average: $\approx 47.61s$ Median: $\approx 48.74s$

Table 6.1: The results of the average and median computation of 10 successful runs for each combination.

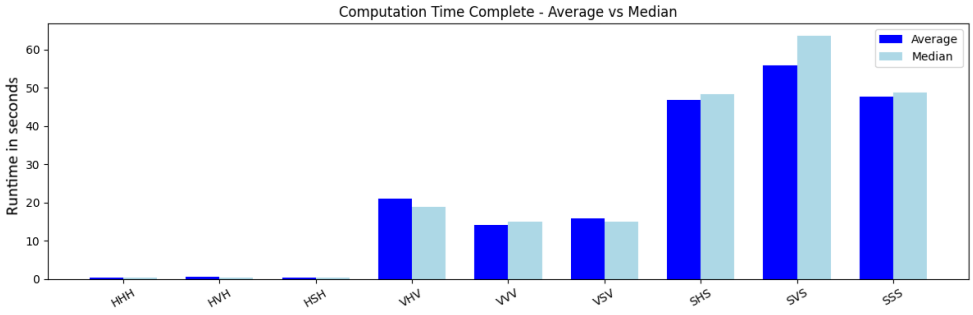


Figure 6.2: The same data as in Table 6.1, visualised.

These results already demonstrate that the current implementation of star sets and $\mathcal{V}$-polytopes are not yet efficient enough to outperform the $\mathcal{H}$-polytopes. However, several factors contribute to this outcome. `HHH` spent the majority of its execution time in the analysis phase. Examining the data from the comparison of the analysis phase in Fig. 6.3 reveals that the median time required for the analysis steps is very similar across all combinations, whereas the average varies significantly. The pronounced peek for `SHS` is caused by a singular outlier, in which the analysis took 20 seconds. This outlier can be seen in Fig. 6.4. Although the result is the correct one, the computation time is a lot higher than expected. This is partly due to the integration phase, which is discussed in more detail later, and partly due to an abnormally high computation time during the analysis phase. The relevant sub steps of the analysis took well under one second. The irregularities of the analysis seem to come from an untraced step in the reachability probability analysis, indicating that there might be a bug present in an improperly measured part of the code. This abnormality has not been yet reproduced and might require further investigation in future work.

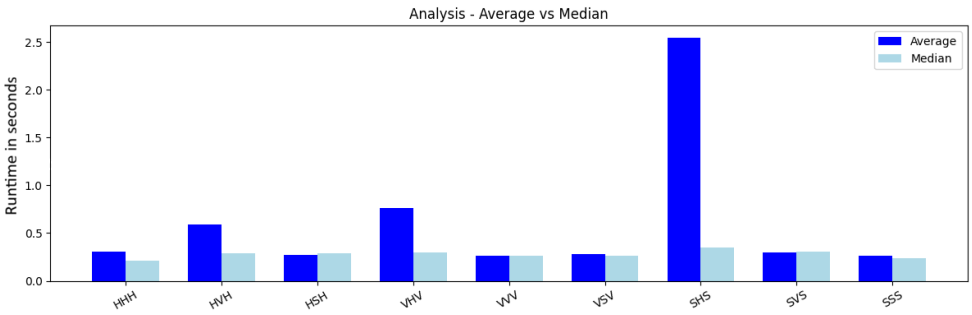


Figure 6.3: Visualisation of the average and median analysis time

```
Probability: 0.842701369572847
Statistical error: 7.306200682064129e-06
Infinity error: 0
[info] 0 Computation Time Complete: 94.9336 s
[info] Counters and timers:
Counters:
Traces: 2
0 Computation Time Complete: 94.9336 s
1 Model creation: 0.000572309 s
2 Analysis: 21.087 s
2.1 Reachability Analysis: 0.0144846 s
2.2 Augment Reach Tree: 0 s
2.3 Filter Segments: 0.0230176 s
2.4 Initialize Trace Structure: 0 s
2.5 Refactor Forward Flowpipe (remove s=0 constraints): 0 s
2.6 Backwards Refinement: 0 s
2.6.2 Compute Intermediate Goal Segment for Backwards Time-
Closure for all Segments: 0 s
2.6.3 Compute Backwards Time Closure for all Segments: 0 s
2.6.4 Perform Backwards Jump for all Segments: 0 s
2.7 Collect segments: 0 s
3 Integration: 73.8459 s
PROBABILITY_INTEGRATION_DIRECT: 73.8459 s
PROBABILITY_INTEGRATION_DIRECT_REMOVE_EMPTY: 0 s
```

Figure 6.4: A run showing a very high computation time for the analysis without any of the sub phases taking longer than usual.

Generally, there seem to be only small differences in regards to efficiency in the analysis phase. Most differences here can be explained by less optimised operations for the $\mathcal{V}$-polytopes and star sets, including frequent conversions to $\mathcal{H}$-polytopes. `RealySt`'s integration process accounts for the main difference in time.

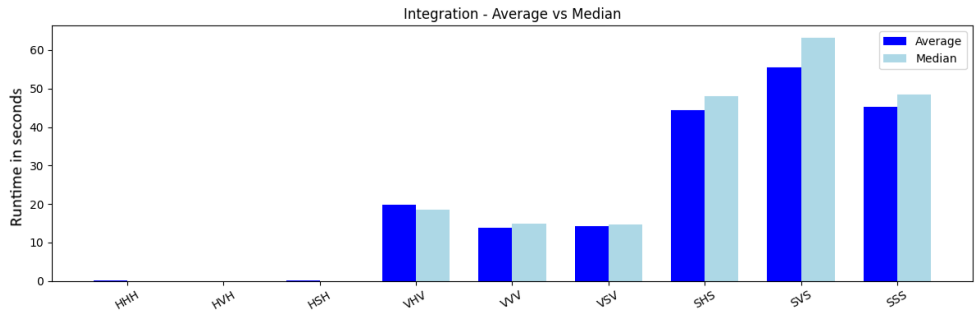


Figure 6.5: Visualisation of the average and median integration time

In Fig. 6.5, it can be seen that the new state set representations are an order of magnitude slower during the integration process, particularly star sets. Moreover, it becomes apparent that, in most cases[1], the homogeneous combination is faster than the other configurations. `HHH` is the fastest of the combinations that use $\mathcal{H}$-polytopes for the integration, while `VVV` is the fastest among the $\mathcal{V}$-polytope integration combinations. This trend can be explained by the additional conversions required during the construction of the stochastic reach tree. The slowdown of the integration process is most likely due to less efficient methods for star sets and $\mathcal{V}$-polytopes and unnecessary conversion, which could be replaced by direct operations in future work.

To further validate this claim, `Callgrind`[Wei08] is used to compare the computations of `HHH` and `SSS` in order to identify the major cost drivers for star sets. `Callgrind` is a profiling tool that simulates programme execution on a virtual CPU, tracking the number of instructions executed, memory access, cache hits, and more. By collecting this detailed information, `Callgrind` enables developers to identify which parts of the code contribute most to runtime and resource consumption. However, because `Callgrind` emulates the programme execution, rather than running it, the profiling process is considerably slower. For SSS, it took 74 times longer than a normal run. For this reason, the analysis in this thesis is limited to `HHH`,`VVV`, and `SSS` as representative examples using only the simple running example.

representations used	`HHH`	`VVV`	`SSS`
instruction count	1.752.936.905	204.207.778.886	909.205.850.960

Table 6.2: The number of instructions used for the probability computation calculated by `Callgrind`

The profiling results show, as expected, that the `HHH` configuration spends the overwhelming majority of its instructions within the reachability analysis itself. In contrast, both `VVV` and `SSS` exhibit a strikingly different performance profile. For these configurations, almost the entire instruction count is consumed by the operations related to the integration phase. These results align perfectly with the observed computational times.

The observed discrepancy in instruction counts (Table 6.2) between the three configurations is considerably more pronounced than the difference in their measured

[1] Especially for $\mathcal{V}$-polytopes for the Integration task.

computational times as seen in Table 6.1. Specifically, `SSS` executes roughly 519 times more instructions than `HHH`, while `VVV` executes around 116 times more instructions than `HHH`. In contrast, `SSS` is approximately 212 times slower than `HHH`, and `VVV` is about 65 times slower than `HHH`[1], showing that the raw instruction volume does not translate linearly into the performance. This can be due to behaviour inside `Callgrinds` virtual CPU, or due to efficiency of the specific operations used.

A closer inspection of the `Callgrind` execution graph in Fig. 6.6 reveals that the `contains` operation is the primary contributor to the excessive instruction count in both `VVV` and `SSS`.

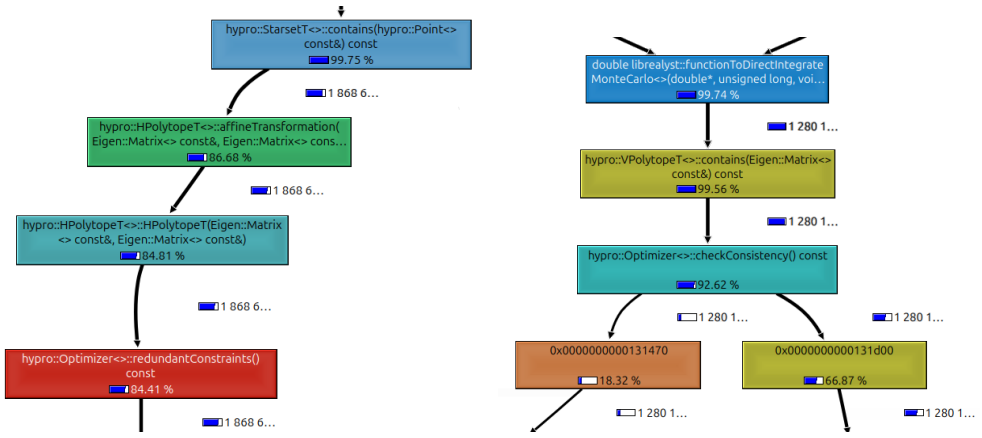


Figure 6.6: A snippet of the `Callgrind` graph, demonstrating the inefficiency of the current integration for `SSS` Fig. 6.6a, and for `VVV` Fig. 6.6b

For star sets, the current implementation of the `contains`[2] operation proceeds by first converting the star set into an $\mathcal{H}$-polytope and then solving the resulting linear programme. In [Tam21], the proposed implementation strategy differs substantially from the approach adopted in this work. Rather than applying an affine transformation to the star set itself, the method transforms the point and subsequently uses the predicate of the star set to formulate the corresponding linear programme. This approach could lead to improved efficiency, as it reduces the number of required transformations.

[1] The median was used for this estimation due to its robustness against outliers.
[2] As provided in 6.1.1.

Implementation 6.1.1 Contains For Star Sets

```
template <typename Number, typename Converter, typename Setting>
bool StarsetT<Number, Converter, Setting>::contains( const Point<
    Number>& point ) const {
    HPolytopeT<Number, Converter, HPolytopeOptimizerCaching>
        transformedStar = this->constraints().
        affineTransformation(mGenerator, mCenter);
    hypro::Optimizer<Number> optimizer( transformedStar.matrix(),
         transformedStar.vector());
    return optimizer.checkPoint(point);
}
```

A potential issue with both approaches is the dimensionality of the star set, as noted in [Vkh24], since the efficiency of linear programmes deteriorates significantly as the dimension increases. While this could, in principle, be mitigated through more extensive use of existing reduction techniques, the current implementation applies reductions each time an $\mathcal{H}$-polytope is created via the `contains` method, as illustrated in Fig. 6.6a. Consequently, when multiple containment checks are performed for a single star set, the resulting polytope is reduced repeatedly, which incurs substantial computational overhead. Simply deactivating the reduction for $\mathcal{H}$-polytopes in the settings does not change anything since the reductions used are hardcoded in the constructors. After manually removing these reductions, the results prove that one major slowdown was due to the reductions. As shown in Fig. 6.7, the total runtime for all configurations employing star sets during integration is significantly reduced compared to the results presented in Fig. 6.2.

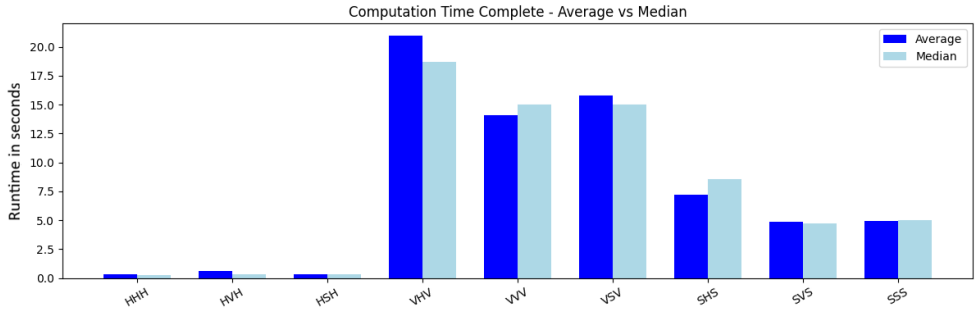


Figure 6.7: Comparison of the last three configurations with reductions manually deactivated.

Further improvements could be made by changing the strategy[1] of the `contains` function or by incorporating reduction steps prior to invoking the `contains` method. As shown in Fig. 6.8, most of the computation time is now spent on solving linear programmes, since, without reductions, these may grow considerably in size.

[1]Instead, a strategy such as the one presented in [Tam21] can be used.

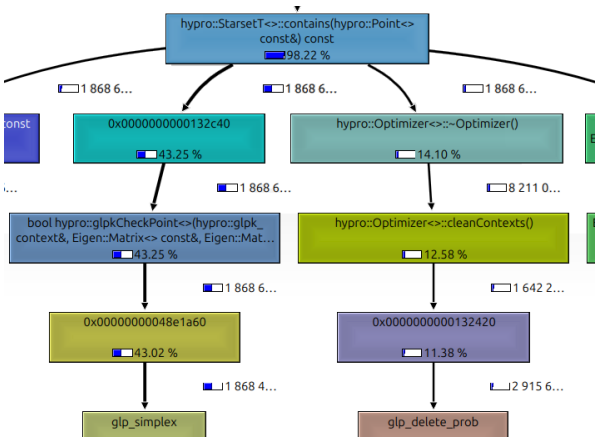


Figure 6.8: A snippet of the `Callgrind` graph generated after removing all reductions.

For $\mathcal{V}$-polytopes, the containment problem is reduced to a linear programme that determines whether a given point can be expressed as a convex combination of the vertices. In this case, applying reduction techniques may likewise be beneficial in alleviating the computational burden of the linear programme. However, since $\mathcal{V}$-polytopes are not the focus of this work, no improvements to their computational efficiency were pursued.

During testing, occasional segmentation faults prevented some runs from completing successfully. Initially, the error was observed in six out of the nine tested combinations. Notably, no failures were observed in runs using star sets in `RealySt`, even after more than 45 executions. This led to the hypothesis that star sets exhibit greater robustness against this error. However, subsequent tests revealed that, although seemingly less frequent, the error also occurs when using star sets in `RealySt`. Tests using older versions of both `RealySt` and `HyPro` also exhibited rare occurrences of the error, leading to the conclusion that it was not introduced by the work carried out in this thesis. Further investigation into the segmentation faults showed that the error occurs during memory reallocation. It may be influenced by the size of the representation used, as well as by the amount of available RAM. This would also explain why the error has not been reported previously, as most researchers are likely to have access to around 32 GB of RAM, rather than the 22 GB allocated to the virtual machine in this setup.

The observed error is not addressed within the scope of this work. Investigations show that it was not introduced by the modifications presented here, as it also occurs in older versions of `RealySt` and `HyPro`. Moreover, the error manifests only very rarely, which makes reliable reproduction difficult and substantially hinders systematic debugging. For these reasons, a thorough analysis and resolution of the issue is beyond the practical scope of this work.

The simple benchmark model shows that, although star sets appeared efficient, their current implementation is not yet sufficient to outperform $\mathcal{H}$-polytopes. However, in isolated analysis steps performed by `RealySt`, star sets seem to demonstrate significant potential in their observed robustness, and even narrowly outperforming $\mathcal{H}$-polytopes in the analysis phase. To investigate this potential further, a second benchmark model was used, in which the analysis phase is significantly more time consuming for all state set representations.

6.2 Car Benchmark

The car benchmark model provided by `RealySt`[DSÁR23] is substantially larger and more complex than the simple running example. The hybrid system models an electric vehicle that initially starts in a charging state. Multiple charging modes are available, depending on the current state of charge of the battery[1]. After the expiration of a random clock, charging terminates and the vehicle begins to drive, even if the battery has not reached full capacity. If a random clock expires while driving, the vehicle arrives at its destination, whereas battery depletion leads to a transition into the empty-battery state. Alternatively, the expiration of another random clock triggers a detour, during which the vehicle may reach a new charging point or deplete its battery. Although the benchmark supports multiple detours, only a single detour is considered in this work, as additional detours incur a substantial computational cost. Furthermore, the system can be modelled either as a singular or a rectangular automaton; in this work, the rectangular automaton is used.

In this setting, the reachability probability computation required, on average, 95.87s for `HHH`, 137.64s for `HVH`, and 94.69s for `HSH`. In this larger model, star sets in `HyPro` again slightly outperformed $\mathcal{H}$-polytopes during the reachability analysis. In all cases, the estimated probability was approximately 0.582, with a statistical error of around 0.00102, which is consistent with the results reported in [DSÁR23].

However, the same drawback observed in section 6.1 persists. Using any representation other than $\mathcal{H}$-polytopes for the integration phase results in a significant slowdown. For instance, when employing $\mathcal{V}$-polytopes for integration, the expected runtime, under the assumption that the slowdown is linear, is around three hours, compared to the approximately 100 seconds when using $\mathcal{H}$-polytopes. After removing the reductions for star sets, as mentioned in section 6.1, it becomes feasible to at least run the benchmark with star sets for `RealySt`. As shown in Fig. 6.9, the overall efficiency of both state set representations is very similar for the larger car benchmark model. While the current implementation of star sets continues to exhibit considerable inefficiency during the integration phase, as illustrated in Fig. 6.10, they remain competitive due to their efficiency in the analysis phase, as shown in Fig. 6.11. This demonstrates that star sets are relevant for future work, especially with larger models. If the inefficiencies of the `contains` method discussed in section 6.1 are addressed, star sets are expected to outperform $\mathcal{H}$-polytopes.

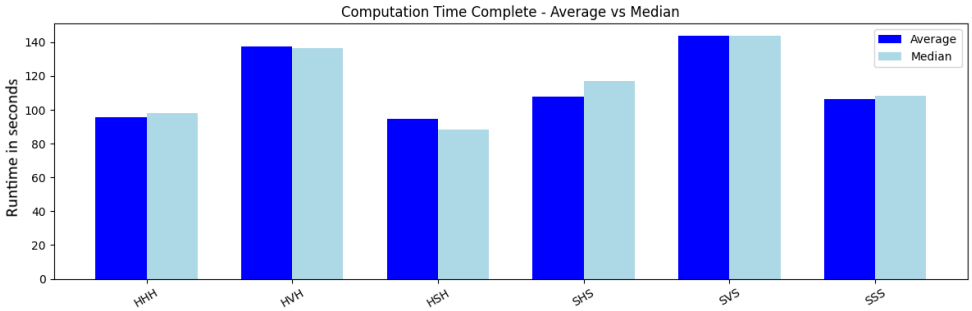


Figure 6.9: A comparison of the use of star sets and $\mathcal{H}$-polytopes in `RealySt` for a larger model.

[1]For this thesis, only two of the three available charging modes are used

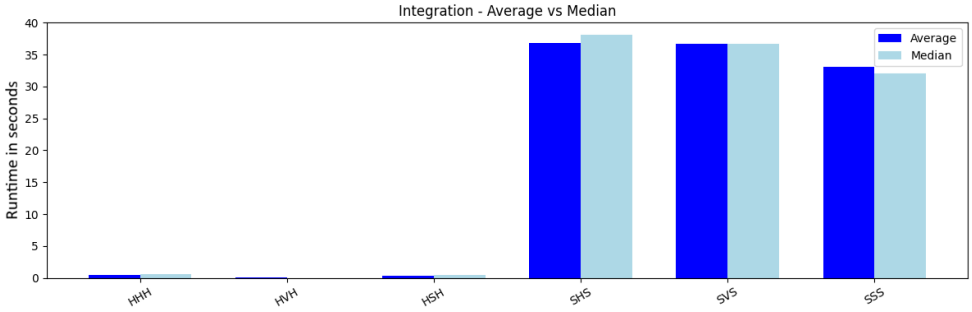


Figure 6.10: Visualisation of the average and median integration time for a larger benchmark.

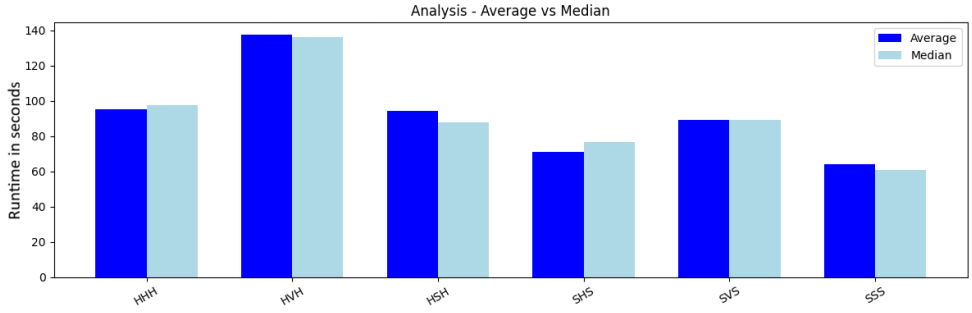


Figure 6.11: Visualisation of the average and median analysis time for a larger benchmark.

Furthermore, out of five runs, `HHH` failed in two cases, `HVH` in one, while `HSH` completed all five successfully. This matches with the prior observation that star sets seem to demonstrate more robustness against the segmentation error found in section 6.1. The higher failure rate of approximately 20% observed for the car benchmark, compared with an estimated 4% for the simple running example, supports the hypothesis that the reallocation error is related to the size of the state set representations and/or the amount of available RAM.

Chapter 7

Conclusion

7.1 Summary

This work has explored the application of star sets as a state set representation for the analysis of stochastic hybrid automata. Theoretical foundations, including hybrid automata, stochastic hybrid automata, reachability, flowpipe construction, and the three state set representations $\mathcal{H}$-polytopes, $\mathcal{V}$-polytopes, and star sets, were introduced. The `HyPro` library and the `RealySt` tool were presented, and their implementations adapted to enable compatibility, allowing `RealySt` to utilise $\mathcal{V}$-polytopes and star sets. The adaptations were evaluated using two benchmark models provided by `RealySt`, and profiling with `Callgrind`. The results indicate that star sets have considerable potential, performing comparably to $\mathcal{H}$-polytopes for larger models due to their efficiency for the analysis. However, the current implementation still exhibits significant inefficiencies, which limit the practicality of star sets during the integration phase.

7.2 Discussion

Despite their promising reputation, the current implementation of star sets faces limitations. While they seem to exhibit robustness against an reallocation error (likely due to a smaller memory footprint) and occasionally outperform $\mathcal{H}$-polytopes in the reachability analysis phase in `HyPro`, their integration performance is an order of magnitude slower than the one of $\mathcal{H}$-polytopes, due to unnecessary reductions. This renders them impractical for larger models, when reductions are applied to $\mathcal{H}$-polytopes. However, after removing these reductions, star sets perform similarly to $\mathcal{H}$-polytopes for the overall probability estimation.

In contrast, $\mathcal{V}$-polytopes face the same issues without offering the corresponding benefits. The evaluation revealed significant variability in runtime, as well as occasional failures, indicating that further optimisation and a more thorough investigation of the implementation are required.

7.3 Future Work

Future research should prioritise improving the efficiency of star set operations, particularly those involved in the integration phase, and those that currently rely on repeated conversions to and from $\mathcal{H}$-polytopes. These conversions impose substantial overhead and prevent star sets from being useful for larger models. The present implementation of `contains` operation exemplifies this problem. Star sets are first converted into $\mathcal{H}$-polytopes and the containment check is then solved as a linear programme. Linear programmes can become highly inefficient, especially with a large amount of constraints. Currently, a reduction for the constraints of $\mathcal{H}$-polytopes is used every time the `contains` operation is called. This is highly inefficient and removal of these reductions brings great improvements. In the future, these reductions should be deactivated for the use of star sets, and instead a dedicated redundancy remover for star sets, which is currently only used for the method `unite`, should be used before containment checks. Furthermore, the strategy to convert the point instead of the star set for the `contains` function, introduced in [Tam21] could be reconsidered and improved upon. Extended benchmarking and profiling could provide deeper insights into the observed failures and runtime variability, enabling targeted optimisations and the fixture of the reallocation error. Finally, exploring alternative state set representations is now feasible and could lead to an improved overall performance and reliability of reachability analysis for stochastic hybrid automata.

Bibliography

[AD14] Matthias Althoff and John M Dolan. Online verification of automated road vehicles using reachability analysis. *IEEE Transactions on Robotics*, 30(4):903–918, 2014.

[AMÁ23] László Antal, Hana Masara, and Erika Ábrahám. Extending neural network verification to a larger family of piece-wise linear activation functions. *arXiv preprint arXiv:2311.10780*, 2023.

[BBHM05] Gerd Behrmann, Ed Brinksma, Martijn Hendriks, and Angelika Mader. Production scheduling by reachability analysis-a case study. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8–pp. IEEE, 2005.

[BD17] Stanley Bak and Parasara Sridhar Duggirala. Simulation-equivalent reachability of large linear systems with inputs. In *International Conference on Computer Aided Verification*, pages 401–420. Springer, 2017.

[BDH96] C Bradford Barber, David P Dobkin, and Hannu Huhdanpaa. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

[BEOB14] Marc Bouissou, Hilding Elmqvist, Martin Otter, and Albert Benveniste. Efficient monte carlo simulation of stochastic hybrid systems. In *The 10th International Modelica Conference 2014*, 2014.

[Che15] Xin Chen. *Reachability analysis of non-linear hybrid systems using taylor models*. PhD thesis, Fachgruppe Informatik, RWTH Aachen University, 2015.

[DDL+12] Alexandre David, Dehui Du, Kim G Larsen, Axel Legay, Marius Mikučionis, Danny Bøgsted Poulsen, and Sean Sedwards. Statistical model checking for stochastic hybrid systems. *arXiv preprint arXiv:1208.3856*, 2012.

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

[DSSR23] Joanna Delicaris, Jonas Stübbe, Stefan Schupp, and Anne Remke. Realyst: A c++ tool for optimizing reachability probabilities in stochastic

hybrid systems. In *EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 170–182. Springer, 2023.

[DV16] Parasara Sridhar Duggirala and Mahesh Viswanathan. Parsimonious, simulation based verification of linear systems. In *International conference on computer aided verification*, pages 477–494. Springer, 2016.

[EP08] Andreas Eidehall and Lars Petersson. Statistical threat assessment for general road scenes using monte carlo sampling. *IEEE Transactions on intelligent transportation systems*, 9(1):137–147, 2008.

[Feh99] Ansgar Fehnker. Scheduling a steel plant with timed automata. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No. PR00306)*, pages 280–286. IEEE, 1999.

[Fre05] Goran Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *International workshop on hybrid systems: computation and control*, pages 258–273. Springer, 2005.

[GDT+02] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Patrick Alken, Michael Booth, Fabrice Rossi, and Rhys Ulerich. *GNU scientific library*. Network Theory Limited Godalming, 2002.

[Gra72] R.L. Graham. An efficient algorith for determining the convex hull of a finite planar set. *Information Processing Letters*, 1(4):132–133, 1972.

[Gra96] Torbjörn Granlund. Gnu mp. *The GNU Multiple Precision Arithmetic Library*, 2(2), 1996.

[HKPV95] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 373–382, 1995.

[HLS00] Jianghai Hu, John Lygeros, and Shankar Sastry. Towards a theory of stochastic hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 160–173. Springer, 2000.

[Hua21] Mengzhe Hua. Approximate model checking for probabilistic rectangular automata with continuous-time probability distributions on jumps, 2021. Bachelor's thesis, RWTH Aachen University, Aachen, Germany.

[hyp] Hypro. `https://github.com/hypro/hypro`. Accessed: 12.12.2025.

[Kan24] Maria Kanzantzi. Reachability analysis for hybrid automata with urgent jumps. Master's thesis, RWTH Aachen University, 2024.

[KJ96] Peter William Kopke Jr. *The theory of rectangular hybrid automata*. PhD thesis, Cornell University, 1996.

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

[LP10] John Lygeros and Maria Prandini. Stochastic hybrid systems: a powerful framework for complex, large scale applications. *European Journal of Control*, 16(6):583–594, 2010.

[Mas23] Hana Masara. Star set-based reachability analysis of neural networks with differing layers and activation functions, 2023. Bachelor's thesis, RWTH Aachen University, Aachen, Germany.

[MHR17] Eike Möhlmann, Willem Hagemann, and Astrid Rakow. Verifying a pi controller using soapbox and stabhyli. In Goran Frehse and Matthias Althoff, editors, *ARCH16. 3rd International Workshop on Applied Verification for Continuous and Hybrid Systems*, volume 43 of *EPiC Series in Computing*, pages 115–125. EasyChair, 2017.

[Mon14] László Monostori. Cyber-physical production systems: Roots, expectations and r&d challenges. *Procedia cirp*, 17:9–13, 2014.

[NDN+16] Johanna Nellen, Kai Driessen, Martin Neuhäußer, Erika Ábrahám, and Benedikt Wolters. Two cegar-based approaches for the safety verification of plc-controlled plants. *Information Systems Frontiers*, 18, 07 2016.

[PSR21] Carina Pilch, Stefan Schupp, and Anne Remke. Optimizing reachability probabilities for a restricted class of stochastic hybrid automata via flowpipe-construction. In *International Conference on Quantitative Evaluation of Systems*, pages 435–456. Springer, 2021.

[rea] Realyst. `https://go.unims.de/RealySt`. Accessed: 12.12.2025.

[RRK10] Derek Riley, Kasandra Riley, and Xenofon Koutsoukos. Reachability analysis of stochastic hybrid systems: A biodiesel production system. *European Journal of Control*, 16(6):609–623, 2010.

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

[SÁW+24] Stefan Schupp, Erika Ábrahám, Md Tawhid Bin Waez, Thomas Rambow, and Zeng Qiu. On the applicability of hybrid systems safety verification tools from the automotive perspective. *International Journal on Software Tools for Technology Transfer*, 26(1):49–78, 2024.

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

[SNÁ17] Stefan Schupp, Johanna Nellen, and Erika Ábrahám. Divide and conquer: variable set separation in hybrid systems reachability analysis. *arXiv preprint arXiv:1707.04851*, 2017.

[Tam21] Dogu Tamgac. Star set representations in the reachability analysis of hybrid systems, 2021. Bachelor's thesis, RWTH Aachen University, Aachen, Germany.

[Tiw07] Hans Raj Tiwary. On the hardness of minkowski addition and related operations. In *Proceedings of the twenty-third annual symposium on Computational geometry*, pages 306–309, 2007.

[TMLM+19] Hoang-Dung Tran, Diago Manzanas Lopez, Patrick Musau, Xiaodong Yang, Luan Viet Nguyen, Weiming Xiang, and Taylor T. Johnson. Star-based reachability analysis of deep neural networks. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 670–686, Cham, 2019. Springer International Publishing.

[Tse20] Phillip Tse. Efficient polyhedral state set representations for hybrid systems reachability analysis. Master's thesis, RWTH Aachen University, Aachen, Germany, 2020.

[Vkh24] Vahe Vkhkryan. A novel reduction method for star sets and its application in neural network verification, 2024. Bachelor's thesis, RWTH Aachen University, Aachen, Germany.

[Wei00] Stefan Weinzierl. Introduction to monte carlo methods. *arXiv preprint hep-ph/0006269*, 2000.

[Wei08] Josef Weidendorfer. Sequential performance analysis with callgrind and kcachegrind. In *Tools for High Performance Computing: Proceedings of the 2nd International Workshop on Parallel Tools for High Performance Computing, July 2008, HLRS, Stuttgart*, pages 93–113. Springer, 2008.

[Zie95] Günter M Ziegler. Lectures on polytopes. *Graduate texts in mathematics*, 152:87–100, 1995.