hybr2d Theory of Hybrid Systems Informatik 2 | RWTH AACHEN UNIVERSITY

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

MASTER OF SCIENCE THESIS

EFFICIENT POLYHEDRAL STATE SET REPRESENTATIONS FOR HYBRID SYSTEMS REACHABILITY ANALYSIS

Phillip Tse

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

Additional Advisor:
Dr. Stefan Schupp

Aachen, 29.4.2020

Abstract

We are surrounded by technical systems which affect our everyday life, some those even with safety-critical responsibilities, for instance the controller of an airbag in a car. For those safety-critical applications, guarantees are needed that certain safety properties are met. By abstracting such systems into a formal model, formal verification techniques can be used to prove the safety of a system. Since the airbag controller of a car is also an example of a system which exhibits both discrete and continuous behaviour, hybrid automata as a modelling formalism which can reflect both types of behavior are often used.
Flowpipe-construction-based reachability analysis can be used to verify safety properties of a hybrid automaton. In this approach, the set of reachable states of the hybrid automaton is overapproximated by sets of geometric shapes (so-called segments); if the intersection of all those shapes with the safety specification is empty, then the system is deemed safe. There exist various representations for segments, whose properties directly influence the speed and the precision of the computation. Previous research has shown that the choice of a state set representation is always a trade-off between precision and running time. To find the right balance between speed and precision, we look at two recently introduced state set representations: template polyhedra and support functions.
In this thesis, we implemented two different approaches to flowpipe-construction-based reachability analysis with template polyhedra: One following the general reachability algorithm for affine hybrid automata using geometric operations, and a Taylor-approximation based approach originally introduced by Sankaranarayanan et al. [SDI08b]. To increase speed and precision of the second method, we implement a policy iteration technique from Sankaranarayanan et al. [SDI08a] that iteratively increases the precision of system invariant conditions. Both approaches as well as the support functions are tested on selected benchmarks and are compared against each other in terms of running time and precision. Our results show that the first approach is in general faster until all optimizations are used on the Taylor approximation based template polyhedra. Additionally we could observe that template polyhedra and support functions were successful on different sets of benchmarks with different structure and properties, exposing a certain duality that needs to be researched further.

Contents

Chapter 1

Introduction

Imagine a satellite needs to be launched from earth into space, and it is the task of some engineers to verify whether the satellite actually reaches the stable distance needed to orbit the earth during its launch. The satellite's launch takes place in two steps: Initially, the rocket containing the satellite will accelerate into the sky as long as its tanks contain fuel. If the tanks are empty, the tanks and the rocket are dropped and the satellite will accelerate, but not as fast as before, by itself until its fuel tanks are also empty. After this two step launching process the satellite must have reached the stable orbit distance or else it will fall back to the earth. Will the satellite reach the stable orbit distance?
The satellite's launch and many other processes we encounter every day are examples of hybrid systems in everyday life, which are systems with both discrete and continuous behaviour. In fact every hybrid system that has a finite number of discrete states (also called locations) with internal continuous behaviour can be described or modelled as a hybrid automaton. For instance the satellite's launch from before can be modelled in this fashion: It has two acceleration locations, one via the rocket fuel tank, one through its own fuel tank. The transition between these two phases is a discrete one, but within each of these phases the acceleration, velocity and height of the satellite change continuously. Often hybrid systems require certain safety properties to be met, especially in safety-critical systems like satellites, planes and trains. Formal verification methods that operate on a model of the hybrid system can validate whether these safety properties are satisfied throughout a whole process. The verification of safety properties in hybrid systems that are modelled via hybrid automata can be executed through a reachability analysis. As in general the exact computation of the reachable state space is undecidable [ACH$^+$95], one technique in reachability analysis is the overapproximation of reachable sets as flowpipes consisting of several convex segments. The segments are geometrically represented in different ways, be it boxes, polytopes or especially important for this thesis, support functions and template polyhedra. Each representation has advantages and disadvantages, which are ultimately a tradeoff between numerical accuracy and computational effort. Since it is not possible to achieve both optimally at the same time, it is desirable to find a representation with the right balance between these two aspects.
This thesis strives to find this balance and for this purpose presents the usage of template polyhedra as a representation for segments in reachability analysis. A template polyhedron as defined in [SDI08b] is a conjunction of inequalities of the form

$\bigwedge_{i=0}^{m} e_i \leq c_i$ where every e_i is a linear expression and $c_i \in (\mathbb{R} \cup \{-\infty,\infty\})$. By fixing the inequalities, which are also called template, while letting c mutable, a family of convex polyhedra can be derived, which can be used to represent segments of different shapes. As a consequence, operations such as intersection and union between template polyhedra of the same family can be efficiently computed in linear time. Under usage of the efficient operations, two flowpipe construction approaches are presented: The first approach follows the standard approach via geometric operations to acquire the flowpipe. The second approach from [SDI08b] uses multiple Taylor approximations to derive new new segments. To further increase the speed and precision of the second approach, we implemented location invariant strengthening [SDI08a] which is a procedure that converges to a tighter bound on the invariants. Both approaches as well as the support functions in conjunction with the template polyhedra and the effects of location invariant strengthening are tested. Other optimizations like fixed point detection are added to obtain an efficient variant of the template polyhedra based reachability analysis.

Structure of the thesis. In order to properly understand all aspects of this topic, a general introduction to hybrid systems analysis is given in Chapter 2, starting with the concept of hybrid automata, which are used to model hybrid systems. With hybrid automata in hand the general reachability procedure will be explained. In this chapter we will also explore support functions as a possible state set representation, their properties, strengths and weaknesses.
After the preliminaries are handled, Chapter 3 will explore template polyhedra more in depth. Different theoretical aspects that occurred during the implementation of the first approach are inspected. We will expand further on the Taylor approximation-based method from Sankaranarayanan et al. [SDI08b] and explore its inner workings for which we will need Lie derivatives, Taylor approximation and dual linear programs, especially for the location invariant strengthening that we will also elucidate on in this chapter.
The fourth chapter presents experimental results for each approach alongside with benchmark results for the support functions coupled with template polyhedra. One section will be dedicated to benchmark results using additional optimizations. The fifth and last chapter concludes this thesis with a summary of our work and gives ideas for future work on template polyhedra.

Related Work. First works on hybrid automata have been made by Alur et al. in [ACH+95], where hybrid automata and their reachability analysis were introduced and found to be undecidable in general. However for some restricted subclasses reachability is decidable. Henzinger [HKPV98] extended this decidability result and found initialized rectangular automata, a subclass of the hybrid automata, to be on the border of decidability.
Since then the flowpipe-based overapproximation of nonlinear continuous behaviour for affine hybrid automata evolved. Initially segments were computed using convex polyhedra [HH94], but with time different state set representations emerged, for instance zonotopes [Gir05], which are the Minkowski sum of several line segments, or of special interest for this thesis, support functions [LGG09]. For the same purpose template polyhedra were proposed. Before that, template polyhedra appeared in static analysis of programs for the generation of invariants for instance in form of the octagon domain of the difference bounds matrices [Min01].

Template polyhedra are still currently being researched on. Dang et al. proposed a method for unbounded time verification that computes abstract semantics of affine hybrid automata with use template polyhedra as state set representations that safely overapproximate the actual semantics [DG11]. This method is based on the abstraction of template polyhedra to have constraints $\bigwedge_i e_i \leq f_i$ where e_i is a linear expression and f_i is a bilinear function [CS11]. Another technique found by Bogomolov et al. follows an counterexample-guided abstract refinement approach to modify the template using spurious counterexamples [BFGH17]. Most related to our work would be the approach of the tool SpaceEx [FLGD$^+$11] that computes segments through a combination of both support functions and template polyhedra, where the more fitting representation is computed when needed.

Chapter 2

Preliminaries

2.1 Notation

As this thesis relies on concepts borrowed from multiple mathematical disciplines, we will first introduce a notation that holds throughout this thesis. Let $\mathbb{N}$ be the set of natural numbers excluding zero and $\mathbb{R}$ the set of real numbers. $\mathbb{R}_+$ denotes the set of positive real numbers excluding zero. Let $d \in \mathbb{N}$ be the dimension. Through lowercase letters a,b,c,.. we denote vectors in $\mathbb{R}^{d\times 1} = \mathbb{R}^d$ until stated otherwise. Especially, let $x \in \mathbb{R}^d$ be the vector of variables. Let i,j,$k \in [0,d-1]$ be indices for a vector $c \in \mathbb{R}^d$ such that c_i denotes the i-th entry in c going from top to bottom. By uppercase letters A,B,C,... we denote matrices in $\mathbb{R}^{m\times n}$ where $m \in \mathbb{N}$ is the number of rows and $n \in \mathbb{N}$ is the number of columns. The number of rows for a matrix A is also notated as $|A| = n$. If an index i,j,k is an index on a matrix A, i.e. A_i, then the i-th row of A is meant, which is a vector in $\mathbb{R}^{1\times n}$. $A_{i,j}$ denotes the entry in the i-th row and the j-th column.

2.2 Hybrid Automata

Before this thesis explains template polyhedra and their properties in detail, it is important to have a clear understanding of the context in which they will be used. This understanding built on the notion of *hybrid systems*.

Hybrid System. A hybrid system is a system whose variables exhibit continuous as well as discrete behaviour [Bra05]. One popular example is the one of the bouncing ball [CSM+15], which will be our running example in the following chapters: In this scenario an elastic ball is dropped from a initial height h_0. During its fall, its height h continuously decreases with its current speed v, while the speed v itself increases with the gravitational acceleration of the earth $g = 9.81\frac{m}{s^2}$. When the ball reaches the ground, it immediately discretely changes its state and bounces upwards, continuously increasing its height h and decreasing its speed v until it reaches the tipping point, from where it falls again according to the initial dynamics.
More advanced examples apart from the one described can be viewed as hybrid systems. Such examples include for instance digital controllers or internet protocols. In order

to prove properties of hybrid systems, it is preferable to have a mathematical model representing them since formal verification methods require a model in order to operate.

Hybrid Automata. For this purpose *hybrid automata* have been introduced [ACH+95]. Formally, a hybrid automaton can be defined as in [Ábr16]:

Definition 2.2.1. *(Hybrid Automaton)*
A hybrid automaton is a tuple $H = (Loc, Var, Edge, Flow, Inv, Init)$ *with*

- *Loc, a set of finitely many locations,*
- *Var, a set of finitely many variables. A valuation* $\nu : Var \to \mathbb{R}$ *is a function assigning each variable a real value. The set of all valuations is* V*.*
- *Edge, a set of finitely many transitions with* $Edge \subseteq Loc \times Lab \times 2^{V^2} \times Loc$*.*
- *Flow, which is a function* $Flow : Loc \to (\mathbb{R}^+ \to V)$ *which maps time-invariant functions to each location. A function* $f(t)$ *is time-invariant when* $f(t) \in Flow(l) \Rightarrow f(t + t') \in Flow(l)$ *for all* $t' \in \mathbb{R}^+$*.*
- *Inv, which is a function* $Inv : Loc \to V$
- *Init, a finite set of initial states. A state is a tuple* $(l,\nu) \in Loc \times V$*. With* Σ *being the set of all states, it is* $Init \subseteq \Sigma$*.*

Intuitively, a hybrid automaton is a finite state machine that has variables $x_0,..,x_{d-1} \in Var$ with continuous dynamics named the flow, which are modelled by time-invariant functions $f(x_0,...,x_{d-1})$ in each location. Usually the flow is denoted by ordinary differential equations (ODEs) of the form $\dot{x_i} = f(x_0,...,x_{d-1})$ for each variable $x_i \in Var$. Only one location can be active at a time, during which the variables change their values according to the flow in each time step. Each location can also have logical constraints over the variables in *Var* called invariants *Inv* that must be satisfied as long as the respective location is active. Through transitions from the transition set *Edge* that are enabled if certain constraints, called the guards, are satisfied, the hybrid automaton can switch from its current active location to a different location from *Loc*, itself included. Taking a transition can potentially trigger a reset, in which variables are set directly to a certain value. Both the guard and the reset can be seen as subsets from the set of valuations V insomuch as only valuations that satisfy the guard can be transformed to another valuation by the reset. Both the locations as well as the transitions and the reset encapsulate the discrete behaviour of a hybrid system, where in contrast the flow in each location models the continuous nature of a hybrid system.
While the definition above defines the syntax of a hybrid automaton, it does not define its semantics. The ideas of time evolution of the variable values according to the flow, satisfying a guard to enable a transition and throughoutly fulfilling the current location invariant can be encapsulated by the operational semantics:

Definition 2.2.2. *(Operational Semantics of a hybrid automaton)*
A hybrid automaton $H = (Loc, Var, Lab, Edge, Flow, Inv, Init)$ *acts according to following rules:*

- *Discrete Rule:*

$$\frac{(l, a, \mu, l') \in Edge \qquad (\nu, \nu') \in \mu \qquad \nu' \in Inv(l')}{(l,\nu) \xrightarrow{a} (l',\nu')}$$

- *Time Rule:*

$$\frac{f \in Flow(l) \wedge f(0) = \nu \wedge f(t) = \nu' \qquad t \geq 0 \wedge \forall 0 \leq t' \leq t : f(t') \in Inv(l)}{(l,\nu) \xrightarrow{t} (l,\nu')}$$

The discrete rule can be interpreted as if the transition from l to l' exists and the valuation change from v to v' is a valid pair while v' satisfies the invariant of l', then we can take the transition. The time rule states that assuming that f is an activity in l and ν, ν' mark the start- and end valuation according to f, then if at all points in time t' between 0 and t it must hold $f(t') \in Inv(l)$. We can define the reachability of a state through the operational semantics: An *execution step* $\rightarrow$ is the taking of a transition according to either rule. A *run* is a sequence of states $\sigma_0 \rightarrow \sigma_1 \rightarrow ...$ with $\sigma_0 = (l_0, \nu_0) \in Init$ and $v_0 \in Inv(l_0)$. A state is *reachable* if there exists a run leading to this state. If a state σ' is reachable from another state σ within finitely many execution steps, we can write $\sigma \rightarrow^* \sigma'$.

Example 2.2.1. *Assume we want to model the bouncing ball hybrid system mentioned before. One possible hybrid automaton modelling this system could be*

$$H = (\{l\},\{x,v\},\{(l,\tau,(G,R),l)\},\{F_x,F_v\},\{x \geq 0\},\{x \in [10.2,10] \wedge v \in [0,0]\})$$

where

- *l is the only location,*
- *x,v are variables modelling the height of the ball and the velocity of the ball,*
- *$G := (x = 0) \wedge (v \leq 0)$ is the guard, which ensures that the transition is only enabled when the ball touches the ground with negative velocity,*
- *$R := (v := -c \cdot v)$ is the reset, which changes the direction of the velocity on impact and reduces it by a fixed constant factor $c \in \mathbb{R}^{\geq 0}$ because energy gets lost during its contact with the floor,*
- *$F_x : \dot{x} = v$ is the flow for the height x, which increases or decreases x continuously with value v according to the laws of physics,*
- *$F_v : \dot{v} = -g$ is the flow for the velocity v, which increases or decreases v the velocity continuously by the gravitational acceleration of the earth $g \approx 9.81 \frac{m}{s^2}$*
- *$x \geq 0$ is the invariant for l. With this we ensure that the height of the ball cannot be negative.*
- *Initially, the ball is at a height of 10 to 10.2 units and is not moving.*

A more intuitive graphical representation of H is the following automaton:

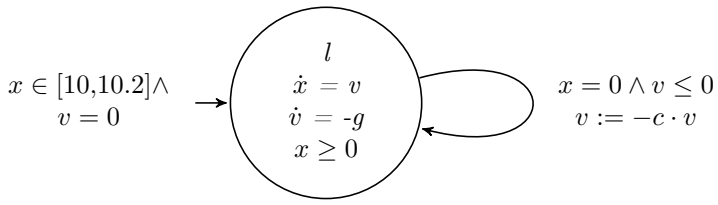


Fig. 1: A hybrid automaton for the bouncing ball system.

Of particular importance for this thesis are the *affine hybrid automata*, which function like general hybrid automata, but are restricted in the shape of the predicates, flows and resets: Only linear inequations are allowed for predicates, all flows must follow affine vector field and resets must be affine transformations. While the flow for each variable can be defined as an linear ordinary differential equation (ODE), we will define it equivalently as an affine vector field to simplify other topics later in the thesis. Note that the trajectories still emit non-linear behaviour. The following definitions follow [SDI08b]:

Definition 2.2.3. *(Vector field)*
A vector field D over $\mathbb{R}^d$ is a function $D : \mathbb{R}^d \to \mathbb{R}^d, x \mapsto D(x)$ mapping a vector $D(x)$ to each point $x \in \mathbb{R}^d$.

Definition 2.2.4. *(Affine Vector Field)*
Given differential equations $\dot{x_i} = f_i(x_0,...,x_{d-1})$, the associated vector field $D(x) = (f_0(x), ..., f_{d-1}(x))$ is affine iff every $f_i(x)$ is an affine function.

Definition 2.2.5. *(Affine Hybrid Automaton)*
A affine hybrid automaton or short AHA is a hybrid automaton H where

- *All $\sigma \in Init$ are defined by conjunctions of linear inequalities*
- *All invariants are defined by conjunctions of linear inequalities*
- *All guards are defined by conjunctions of linear inequalities*
- *All resets are affine transformations of the form $x := Ax + b$ for $A \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$*
- *All flows are affine vector fields.*

2.3 General Reachability Analysis

After defining hybrid automata we can proceed to dive into the formal verification of properties using hybrid automata. For this, recall the bouncing ball model from before: Would it ever be possible for the ball to reach a height that is greater than 10.2 units? The answer is no since the ball loses energy with each bounce and therefore cannot reach a greater maximum height after a bounce. While this example question is still solvable through intuition, its abstraction leads to a important question in hybrid systems verification: Given a hybrid automaton H with the initial set *Init*, which states are reachable from *Init* under H? More importantly, additionally given a set of bad states $Bad \subseteq \Sigma$, which can model unwanted properties in the system, is it possible to avoid all states from *Bad*? If it is not possible, is it at least possible to

avoid the states from Bad at least for a fixed amount of time T? To answer these questions, we will first define the problem according to [Ábr16]:

Definition 2.3.1. *(Forward Reachability Problem)*
Given a hybrid automaton H with initial set $Init$, the forward reachability problem is the problem to compute the set of all states that are reachable from $Init$:

$$Reach(Init)^+ = \{\sigma' \in \Sigma \mid \exists \sigma \in Init : \sigma \to^* \sigma'\}$$

Definition 2.3.2. *(Forward Reachability Analysis)*
Given a hybrid automaton H with initial set $Init$ and a set of bad states $Bad \subseteq \Sigma$, the forward reachability analysis checks whether

$$Reach(Init)^+ \cap Bad = \emptyset.$$

The general algorithm to compute $Reach(Init)^+$ starts with $Init$ and then checks which states are reachable from all $\sigma \in Init$. As long as new states have been reached, we can add these new states to $Reach(Init)^+$ and can then check for each new state which other states are reachable from there. If no more new states are found, the computation is complete and the result can be returned. The algorithm is summarized in Algorithm 1.

Input: Initial States $Init \subseteq \Sigma$
Output: Set of all reachable states $Reach(Init)^+$
  $R := \emptyset$
  $R_{\text{new}} := Init$
  while $R_{\text{new}} \neq \emptyset$ **do**
    $R := R \cup R_{\text{new}}$
    $R_{\text{new}} := \text{Reach}(R_{\text{new}}) \setminus R$
  end while
  return R

Algorithm 1: General Forward Reachability Algorithm.

Backward Reachability. As the name suggests, there is not only a forward reachability problem, but also a backward reachability problem [ACH+95] which is the problem to compute $Reach(B)^- = \{\sigma' \in \Sigma \mid \exists \sigma \in B : \sigma' \to^* \sigma\}$ for a given set of states $B \subseteq \Sigma$. Given the set of bad states Bad, we can compute whether there exists a run from some initial state to some state in Bad. If that is the case then a counterexample has been found and the property modelled by the bad states holds. In this thesis we will only consider forward reachability algorithms.

Decidability. In general, the forward reachability problem for general hybrid automata is undecidable [ACH+95] since it is not possible to analytically compute $\text{Reach}(R_{\text{new}})$. However, for several subclasses of hybrid automata, for instance the timed automata where all variables have slope one [AD94], or the initialized rectangular automata [HKPV98], the forward reachability problem is decidable. We will study the reachability analysis of affine hybrid automata for whom the forward reachability problem is undecidable. Since the forward reachability analysis is undecidable for this kind of hybrid automaton, we need to approximate all sets of reachable states.

An overapproximation is used if we want to prove that the reachable sets do not intersect the bad states and therefore provide safety. An underapproximation is used to guarantee that the system definitely has a run to a bad state. Since in hybrid systems verification we mostly want to show that the modelled systems cannot reach a given set of bad states, an overapproximation algorithm is used.

Flowpipe Construction. One class of these algorithms are the flowpipe construction based algorithms, which divide the time bound $T \in \mathbb{R}$ into $N \in \mathbb{N}^{>0}$ time steps $\delta = \frac{T}{N}$ such that we have time segments $[0,\delta]$,$[\delta,2\delta]$,...,$[(N-1)\delta,T]$. For each time segment $[(i-1)\delta, i\delta]$ we wish to compute the overapproximation of the reachable states in this time interval. For an affine hybrid automaton, the flow of every variable x_i is of the form $\dot{x_i} = f(x_0,...,x_{d-1})$ where $f(x_0,...,x_{d-1})$ is an affine function by definition. In $\mathbb{R}^d$ with an euclidean metric, which is the space the flow computation is located in, all affine functions imply *Lipschitz continuity* [For84]. Lipschitz continuity implies that a unique solution $x(t,x_0)$ to $\dot{x_i} = f(x_0,...,x_{d-1})$ can be found starting from an initial value x_0 after a certain time t [For84]. We use this to define the reachable states:

Definition 2.3.3. *(Reachable States)*
Let $X_0 \subseteq \Sigma$ be the initial states. The set of reachable states $\mathcal{R}_{[t,t']}(X_0)$ reachable from X_0 at time interval $[t,t']$ is defined as [Ábr16]:

$$\mathcal{R}_{[t,t']}(X_0) = \{x_s \in \Sigma \mid \exists x_0 \in X_0.\ \exists t \leq s \leq t'.\ x_s = x(s,x_0)\}$$

The reachable states are all valuations x_s that are the reachable from x_0 via the flow in the time interval $[t,t']$. The set of all reachable states within the time bound T is therefore $\mathcal{R}_{[0,T]}(X_0)$. We can overapproximate $\mathcal{R}_{[0,T]}(X_0)$ by overapproximating each actual segment $\mathcal{R}_{[(i-1)\delta,i\delta]}(X_0)$ through a *polyhedral overapproximation* $\hat{\mathcal{R}}_{[(i-1)\delta,i\delta]}$. The whole flowpipe approximation is then the union of all overapproximating segments:

$$\hat{\mathcal{R}}_{[0,T]} = \bigcup_{i=1}^{N} \hat{\mathcal{R}}_{[(i-1)\delta,i\delta]}$$

The set of all reachable flowpipes form an overapproximative set of $Reach(Init)^+$. During the flowpipe construction the following definitions are needed:

Definition 2.3.4. *(Convex Set)*
A set $S \subseteq \mathbb{R}^d$ with $d \in \mathbb{N}$ is convex [BV04] if it holds

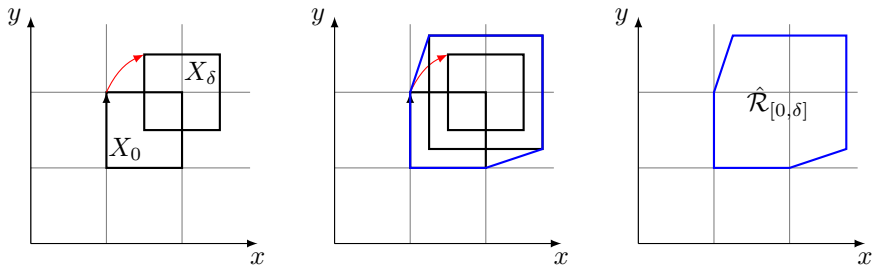
$$\forall s,r \in S\ \forall \lambda \in [0,1] : \lambda s + (1-\lambda) r \in S.$$

A set is convex if the line segment between two points s and r in S is also completely contained in S.

Definition 2.3.5. *(Convex Hull)*
The convex hull $conv(S)$ is defined as [BV04]

$$conv(X) = \left\{ \sum_{i=1}^{n} \lambda_i x_i \mid \sum_{i=1}^{n} \lambda_i = 1 \wedge \lambda_i \geq 0 \wedge x_i \in X \right\}.$$

The convex hull is the smallest convex set that contains X.



(a) (b) (c)

Figure 2.1: **(a)** Initial set X_0 and its transformed set $X_\delta = e^{A\delta}X_0$. The red arrow is not included nonlinear behaviour that needs to be included. **(b)** X_δ is bloated with a small box B such that the convex hull of X_0 and the bloated set contain the red arrow. **(c)** The resulting overapproximating first segment $\hat{\mathcal{R}}_{[0,\delta]}$.

Definition 2.3.6. *(Minkowski Sum)*
The Minkowski sum [Zie12] of two sets $A \subseteq \mathbb{R}^d$ and $B \subseteq \mathbb{R}^d$ is defined as

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

This can be geometrically interpreted as the set that results from moving B along the border of A.

First Segment Computation. To compute the first segment $\hat{\mathcal{R}}_{[0,\delta]}$ given the initial set X_0 the following operations need to be done: At first, X_0 is transformed according to the flow. Since the flow of an affine hybrid automaton is an affine vector field where every variable x_i follows the linear ordinary differential equation $\dot{x_i} = f(x_0,...,x_{d-1})$, we can approximate the linear ODE for the next valuation vector $x(t,x_{\text{init}})$ via matrix exponentiation:

$$x(t,x_{\text{init}}) = e^{A\delta}x_{\text{init}} = \sum_{k=0}^{\infty} \frac{(A\delta)^k}{k!}x_{\text{init}}$$

Generalizing that to state sets yields:

$$X_\delta = e^{A\delta}X_0$$

Geometrically, X_0 is translated, scaled and rotated to X_δ according to the flow.
At this point, it could be possible to compute the convex hull of X_0 and X_δ to get the first segment. But doing so would exclude possible nonlinear trajectories arising from the ODE. To include these trajectories, X_δ is bloated with a box B via the Minkowski sum, just big enough for the convex hull of X_0 and $X_\delta \oplus B$ to include all of them. The detailed computation of the bloating box is described in [LG09]. The whole process of computing the first segment is depicted in Figure 2.1. The overall first segment covering all possible trajectories within time interval $[0,\delta]$ is:

$$\hat{\mathcal{R}}_{[0,\delta]} = conv(X_0 \cup (e^{A\delta}X_0 \oplus B))$$

Time Evolution. From the first segment $\hat{\mathcal{R}}_{[0,\delta]}$ we can derive the next segment $\hat{\mathcal{R}}_{[\delta,2\delta]}$ by applying matrix exponentiation on $\hat{\mathcal{R}}_{[0,\delta]}$. Since $\hat{\mathcal{R}}_{[0,\delta]}$ is already including

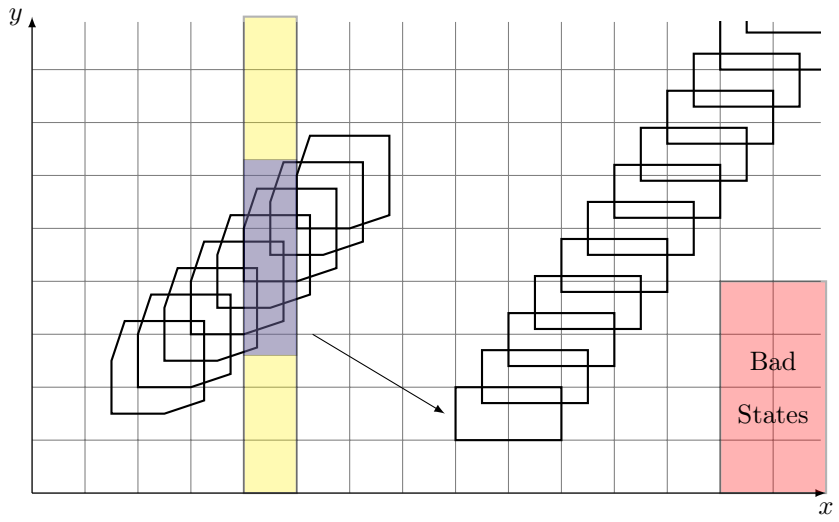


Figure 2.2: Affine hybrid automaton reachability analysis. The first flowpipe intersects the guard (yellow). The sets are aggregated (blue) and reset is applied (arrow). The second flowpipe is computed from there. No segment intersects the bad states.

all possible trajectories, it is guaranteed that $\hat{\mathcal{R}}_{[\delta,2\delta]}$ remains overapproximative. This process can be repeated as long as the current flowpipe does not intersect bad states, invalidate the invariant or reach the time bound T.
Every segment that gets computed needs to be checked whether there exists a state in it that satisfies the invariant via the set intersection operator. If that is the case the computation can continue with the satisfying states, in the other case the flowpipe construction of the current flowpipe can be terminated. After that, the segment is intersected with the bad states to check validate safety. If the intersection is not empty then bad states were hit and safety could not be verified, else the computation can continue. Then the current segment needs to be intersected with the guards of all outgoing transitions of the current location to see whether any of them are enabled. An enabled guard induces that a transition in the affine hybrid automaton can be taken. All states in the current segment that satisfy the guard are transformed according to the reset and form the initial set of a new flowpipe where the flowpipe construction is started anew. As every segment that intersects a guard would start a new flowpipe, which in turn starts a new flowpipe for every segment that satisfies another guard, an *aggregation method* can be used to reduce the exponential blowup of flowpipes [Sch19]. Instead of creating a flowpipe for every segment that intersects a guard G, all segments that satisfy G can be unified via the convex hull to one segment to only create one flowpipe. Although aggregation introduces additional overapproximation error, it can be used when computational speed is a priority [Sch19]. A whole overview of the reachability analysis is depicted in figure 2.2.

State Set Representations. As depicted in the provided figures, states can be described by convex geometric objects. For computational simplicity it is advantageous to have one state set representation for all segments. A state set representation must be able to carry out the following operations that are needed for the reachability

analysis:

- Affine transformation $Ax + b$ for resets and time evolution
- Minkowski Sum $\oplus$ for bloating
- Intersection $\cap$ for intersection invariants, bad states and guards
- Union $\cup$ for uniting X_0 and X_δ to one set get the convex hull the union
- Check for emptiness to check whether the intersected sets are empty or not
- Membership of a segment in another segment for fixed point detection

Some representations of convex sets most include *boxes* and *convex polytopes.*

Definition 2.3.7. *(Box)*
Let $[l,u]$ *be an interval in* $\mathbb{R}$ *for a lower bound* l *and upper bound* u *such that* $l \leq u$. *Let* $I = \big([l_0,u_0] \dots [l_{d-1},u_{d-1}]\big)$ *be a vector of intervals in* $\mathbb{R}^d$. *A box* $\mathcal{B} \subseteq \mathbb{R}^d$ *is defined as [Sch19]:*

$$\mathcal{B} = \{x \in \mathbb{R}^d \mid \forall 0 \leq i \leq d-1.\ x_i \in [l_i,u_i]\}$$

Assume we use boxes as state set representations. It is known that boxes have efficient algorithms for every listed operation needed for the flowpipe construction, which makes them a very efficient representation with regards to the running time provided the operations maintain closure. Their downside lies in the overapproximation error that appears due to the need of maintaining closure: A state set which is not a box will be overapproximated by a box and will introduce a large overapproximation error. This error can then be scaled up by the time evolution, as the transformed error must again be overapproximated; the overapproximation error increases from one segment to the next. This effect is known as the *wrapping effect* [LGG09]. On the other side we have convex polytopes in halfspace representation:

Definition 2.3.8. *(H-Polyhedron)*
A closed halfspace in $\mathbb{R}^d$ *given a normal vector* $a \in \mathbb{R}^d$ *and an offset* $b \in \mathbb{R}$ *is the set* $\mathcal{H} := \{x \in \mathbb{R}^d \mid a^T \cdot x \leq b\}$. *A H-Polyhedron* $\mathcal{P}$ *is an intersection of* m *closed halfspaces* $\mathcal{H}_i$ *[Zie12]:*

$$\mathcal{P} := \bigcap_{i=0}^{m} \mathcal{H}_i$$

If P is bounded, then it is called a *H-polytope.* P can be written as $P = Ax \leq b$ with $A \in \mathbb{R}^{m \times d}$ and $b \in \mathbb{R}^d$ where every row i in A and b corresponds to the normal and the offset of $\mathcal{H}_i$. H-polytopes are more precise in approximating a convex set than the boxes as the halfspaces. The difference in approximation precision can be seen in Figure 2.3. Additionally, H-polytopes are efficient at the operation intersection as the intersection of two H-polytopes $\mathcal{P}$ and $\mathcal{P}'$ is the conjunction of the halfspace constraints of both H-polytopes. But for other operations like the check for emptiness, *linear programs (short LPs)* must be solved, which need polynomial time for instance via interior point methods [DH12], though in practice the simplex method is often times faster although it has a exponential worst case running time complexity [PS98]. The largest effect on the running time have the affine transformation, Minkowski sum

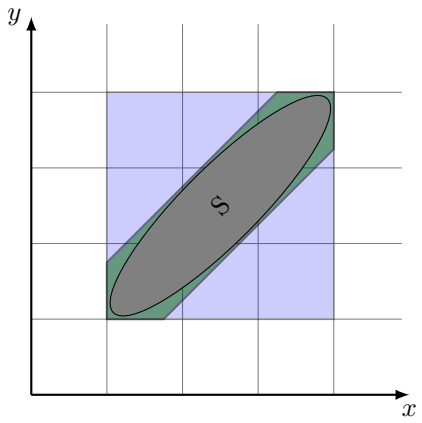


Figure 2.3: A convex set S and its overapproximation by a box (blue) and a H-polytope (green).

and union operations: Computing these operations on H-polytopes needs exponential time [Sch19]. Thus, the H-polytope is precise but slow. In general it holds that every representation is a trade-off between speed and precision.

2.4 Support Functions

Support functions are an efficient state set representation first presented for hybrid systems reachability analysis in [LGG09]. Support functions are structures that enable efficient operations on convex sets by modifying support values in given directions according to the operations. Each support value then corresponds to the support of a supporting hyperplane of the underlying set. Analogous to the idea of H-polytopes, which represent a set as the intersection of finitely many closed halfspaces, a convex set can also be polyhedrally overapproximated by computing the support, or shorter evaluating the set in finitely many directions. Figure 2.4 depicts the idea.

Definition 2.4.1. *(Support Value)*
The support value [LGG09] of a set $S \subseteq \mathbb{R}^d$ for a given direction $l \in \mathbb{R}^d$ is defined as

$$\rho_S(l) = \max_{x \in S} l \cdot x$$

which is the support value of the supporting hyperplane $l^T x \leq \rho_S(l)$ to S.

Proposition 2.4.1. *A convex set $S \subseteq \mathbb{R}^d$ is uniquely determined by its support functions in all directions:*

$$S = \bigcap_{l \in \mathbb{R}^d} \{x \in \mathbb{R}^d \mid l \cdot x \leq \rho_S(l)\}$$

One of the most important properties of support functions are the rules for inferring support functions from after the application of an operation using support functions from before the operation. This is an important difference to the H-polytopes. These rules establish an efficient way of symbolically executing all needed operations:

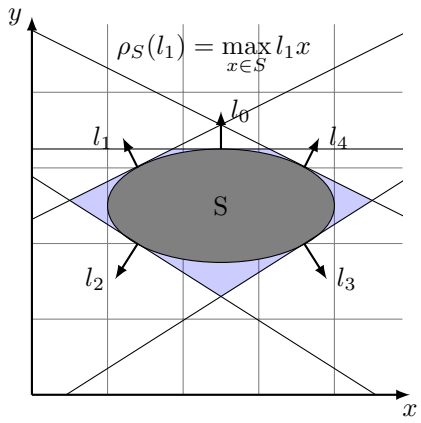


Figure 2.4: A convex set S is overapproximated by the intersection of five halfspaces built from support halfspaces. Each direction l_0,...,l_4 corresponds to a supporting hyperplane $\mathcal{H}_i : l_i \cdot x = \rho_S(l_i)$.

Theorem 2.4.2. *Let S,$S' \subseteq \mathbb{R}^d$ be convex sets. Let $M \in \mathbb{R}^{m \times d}$ be a matrix, $l \in \mathbb{R}^d$ and $l \in \mathbb{R}^m$ be vectors of different dimensions and $\lambda \in \mathbb{R}^+$ be a positive scalar. It then holds:*

$$\begin{aligned}
\rho_{MS}(l) &= \rho_S(M^T l) \\
\rho_{MS+b}(l) &= \rho_S(M^T l) + l \cdot b \\
\rho_{\lambda S}(l) &= \rho_S(\lambda l) = \lambda \rho_S(l) \\
\rho_{conv(S \cup S')}(l) &= \max(\rho_S(l), \rho_{S'}(l)) \\
\rho_{S \oplus S'}(l) &= \rho_S(l) + \rho_{S'}(l) \\
\rho_{S \cap S'}(l) &\leq \min(\rho_S(l), \rho_{S'}(l))
\end{aligned}$$

All operations except the last one are exact equations. There is an exact equation for $\rho_{S \cap S'}(l)$ according to [LGG09], but its equation is computationally expensive. The denoted inequation on the other hand is a valid overapproximation of $\rho_{S \cap S'}(l)$ and is fast to compute.
If we now for instance compute the support for $\rho_{MS \oplus S'}(l) = \rho_{MS}(l) + \rho_{S'}(l) = \rho_S(M^T l) + \rho_{S'}(l)$, one can observe that at the end $\rho_{MS \oplus S'}(l)$ is only dependent on $\rho_S(M^T l)$ and $\rho_{S'}(l)$, which lastly are maximization queries for S and S' in some direction l. Since this process can be repeated for every rule, one can depict a support function as a tree. A node in this tree is either an operation or a state set; all leaves must be state sets and all non-leaves must be operations. Each node points to the nodes where it gets its support function arguments from. If a support function is queried, the corresponding node is queried and that node queries its children, who query their children and so forth. When the query reaches a leaf, it computes the support value via a maximization in the given direction and returns this support value to its parent. That parents aggregates all needed support values and then returns its result to its parents until the queried node can return the requested support value. So in order to get a support value, one must traverse the whole tree. The traversal is pictured in Figure 2.5.

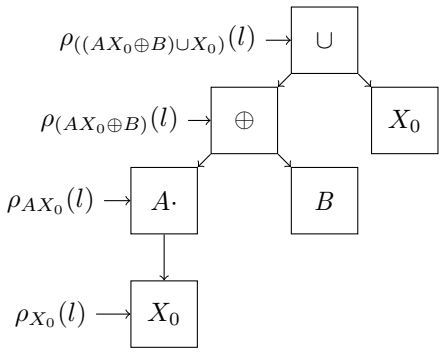


Figure 2.5: Computation of the first segment using support functions. At each node the support value on the left side of the arrow can be queried.

Example 2.4.1. *We will explain the traversal using Figure 2.5 as an example. Figure 2.5 demonstrates the support function tree that is build for the computation of the first segment.*
Assuming we want to query the support value for the first segment $\rho_{((AX_0 \oplus B) \cup X_0)}(l)$ *in direction* l *starting at the node* $\cup$*, we will conduct the traversal by descending to all the leaves, computing a support value, and aggregating the values needed for* $\rho_{((AX_0 \oplus B) \cup X_0)}(l)$*.*
Starting with $\rho_{((AX_0 \oplus B) \cup X_0)}(l)$*, we can instead compute* $\max(\rho_{(AX_0 \oplus B)}(l), \rho_{X_0}(l))$ *for which we need* $\rho_{(AX_0 \oplus B)}(l)$ *and* $\rho_{X_0}(l)$*. We can obtain* $\rho_{X_0}(l)$ *by making a maximization query for* X_0 *in direction* l*, which can be a LP with cost function* $l^T x$*.*
For $\rho_{(AX_0 \oplus B)}(l)$ *we need to query the children of* $\oplus$*, which in turn query their children. When* $\oplus$ *receives* $\rho_{AX_0}(l)$ *and* $\rho_B(l)$ *from its children both support values can be aggregated by* $\rho_{(AX_0 \oplus B)}(l) = \rho_{AX_0}(l) + \rho_B(l)$*.*
This support value can then be returned to the parent node $\cup$ *which can then return the wanted support value.*

Support functions are heavily reliant on the maximization capabilities of the underlying representation S. If S is efficient at maximizing into any direction, then all operations carried out on the support function benefit from that. The template polyhedra presented in the next section are one such class.
The support functions suffer from one weakness: The running time for checking emptiness of two intersecting non-empty support functions. This operation can be executed by computing the support values $\rho_A(l_i)$ of the first set A as well as $\rho_B(-l_i)$ of the second set B in octagonal directions $\{l_0,...,l_k\}$ with $k \in \mathbb{N}$. If $\rho_A(l_i) \leq -\rho_B(-l_i)$ for at least one direction l_i then A intersects B and the intersection cannot be empty. Checking emptiness with this algorithm requires $2k$ traversals for each support function computed and is therefore costly.

Chapter 3

Template Polyhedra

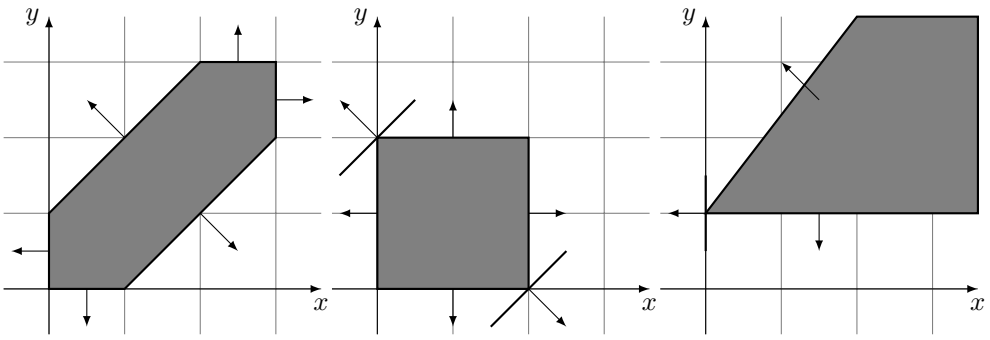
This chapter discusses the main state set representation of this thesis, namely the *template polyhedra*. First introduced to hybrid systems reachability analysis by Sankaranarayanan et al. in [SDI08b] template polyhedra are closely related to H-polytopes, as they can also be represented by the intersection of finitely many closed halfspaces. But in contrast to H-polytopes the main idea of template polyhedra is to fix the constraint matrix, such that only the offset vector can differ. This allows for several simplifications of many binary operations, as only the offset vectors need to be compared, which can be done in linear time in the number of constraints.
At first, we will formally define template polyhedra and explore their properties more in detail. Afterwards, we will present an implementation made with the *HyPro* library, a C++ library for flowpipe-based reachability analysis of affine hybrid automata [SÁBMK17], and explain some optimizations made therein in Section 3.2. Section 3.3 looks at a different approach regarding the reachability analysis which was proposed by Sankaranarayanan et al. in [SDI08b]. This approach relies on the concept of Lie derivatives and Taylor approximation, which will be explained beforehand. Especially the first segment computation and the time evolution will be unfolded. In the last Section 3.4 techniques for optimizing the Sankaranarayanan reachability analysis are presented, the most important one being the *location invariant strengthening*.

3.1 Definition and Properties

A linear expression $e(x)$ over a variable vector $x \in \mathbb{R}^d$ is an expression of the form $e(x) = \sum_{i=0}^{d-1} a_i x_i$ where $a_0,...,a_{d-1} \in \mathbb{R}$. We can then define:

Definition 3.1.1. *(Template)*
A template [SDI08b] is a set of linear expressions $H = \{h_0(x),...,h_{m-1}(x)\}$ with $m \in \mathbb{N}$ over a variable vector $x \in \mathbb{R}^d$.

We represent a template by a matrix H in $\mathbb{R}^{m \times d}$ such that the i-th row in H corresponds to the coefficients of the i-th linear expression. The linear expression of one row would be $H_i x = h_{i0}x_0 + h_{i1}x_1 + ... + h_{id}x_d$.



(a) (b) (c)

Figure 3.1: Template polyhedra for Example 3.1.1 **(a)** A bounded template polyhedron. **(b)** A template polyhedron using the same template, but two constraints are redundant. **(c)** An unbounded template polyhedron using the same template.

Definition 3.1.2. *(Template Polyhedron)*
A template polyhedron [SDI08b] over a fixed template H is a H-polyhedron of the form $Hx \leq c$, or short $\langle H,c\rangle$, for $c \in (\mathbb{R} \cup \{-\infty,\infty\})^d$.

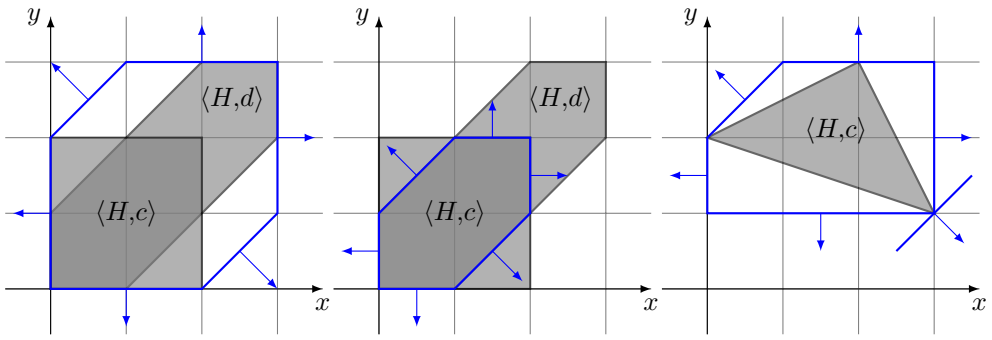
A template H induces a family of template polyhedra that have the same constraints H, but use different offset vectors c. Some constellations of coefficients can make certain halfspaces redundant, while the introduction of $-\infty$ and ∞ into the domain of c adds possible unboundedness to the template polyhedra. Note that the right choice of coefficients can also make a template polyhedron empty.

Example 3.1.1. *Let us look at Figure 3.1. All three template polyhedra are from the same family with the template:*

$$H = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix} \text{. (a) } \langle H, \begin{pmatrix} 3 \\ 0 \\ 3 \\ 0 \\ 1 \\ 1 \end{pmatrix} \rangle, \text{ (b) } \langle H, \begin{pmatrix} 2 \\ 0 \\ 2 \\ 0 \\ 2 \\ 2 \end{pmatrix} \rangle, \text{ (c) } \langle H, \begin{pmatrix} 0 \\ \infty \\ \infty \\ -1 \\ 1 \\ \infty \end{pmatrix} \rangle$$

The middle template polyhedron (b) is an example of possibly redundant constraints. Since the constraints are in the template, they still need to be considered for every operation. The right template polyhedron (c) is unbounded as some of its offset values are infinity.

Properties. If we use template polyhedra for overapproximating sets, then it is algorithmically desirable to have a unique representation under a template H for every possible set. This property is given by the smallest overapproximating template polyhedron of a given set S and can be computed by determining the support value in each template row direction H_i by the maximization query $\max H_i x$ *subject to* $x \in S$. If S is a convex polytope, then every maximization query can be solved by a linear program (short LP).



(a) Union (b) Intersection (c) Overapproximation

Figure 3.2: Operations on template polyhedra from Figure 3.1. **(a)** The blue shape denotes $\langle H,c\rangle \cup \langle H,d\rangle$ **(b)** The blue shape denotes $\langle H,c\rangle \cap \langle H,d\rangle$ **(c)** The blue shape denotes the overapproximation of $\langle H,c\rangle$ with template H from Example 3.1.1. Note the approximation error which are the white regions within the blue shape.

Lemma 3.1.1. *(Uniqueness) Let $\alpha_H(S)$ be the offset vector of the least overapproximating template polyhedron with template H for a closed set S. Then $\langle H, \alpha_H(S)\rangle$ is unique for S.*

$\alpha_H(S)$ is then called canonical. A template polyhedron $\langle H, c\rangle$ where $c = \alpha_H(S)$ is also called canonical. The uniqueness of $\langle H, \alpha_H(S)\rangle$ also holds when H contains redundant constraints, as they also have a least overapproximating support value in their respective direction.
For the next properties, let us assume the vectors $c,d,e,f \in (\mathbb{R} \cup \{-\infty,\infty\})^d$ are canonical. Let us define that $c \leq d \iff c_i \leq d_i \ \forall i \in [0,d-1]$ and analogously $c = d \iff c_i = d_i \ \forall i \in [0,d-1]$ as well as for all $i \in [0,d-1]$ it holds $coeffmax(c,d) = e$ with $e_i = \max(c_i,d_i)$ and $coeffmin(c,d) = f$ with $f_i = \min(c_i,d_i)$. Then the following holds [SDI08b]:

Lemma 3.1.2. *(Efficient Operations on canonical template polyhedra)*

- *Subset:* $\langle H, c\rangle \subseteq \langle H, d\rangle \iff c \leq d$
- *Equality:* $\langle H, c\rangle = \langle H, d\rangle \iff c = d$
- *Union:* $\langle H, c\rangle \cup \langle H, d\rangle = \langle H, coeffmax(c,d)\rangle$
- *Intersection:* $\langle H, c\rangle \cap \langle H, d\rangle = \langle H, coeffmin(c,d)\rangle$
- *Support in direction l: If the maximization direction $l \in \mathbb{R}^d$ is a template constraint in H at row i, then return c_i, else make an LP call* $\max l^T x$ *subject to* $x \in \langle H,c\rangle$.

Most of the operations can be performed in time linear in the number of template rows. An exception to this is the maximization operation: If the maximization direction is a template constraint, then we can directly return the corresponding offset value. This is more efficient than a LP call, which can be solved in polynomialial time. Since H-polytopes have to make an LP call in every case for this operation, template polyhedra cannot be slower than H-polytopes regarding maximization. From this observation we

can deduce that the choice of the template is crucial for the computation speed during the reachability analysis. This also holds for the precision: Since all efficiency of the operations stems from both operands having the same template, a state set S that does not conform to a given template H must be overapproximated by H, which, depending on the template, can introduce a lot of approximation error. The overapproximation of a set not represented by the template H itself can be accomplished as described before, by solving the LP $\max H_i x$ *subject to* $x \in S$ for every row H_i. A depiction of the operations can be viewed in Figure 3.2, the approximation error is visible there, too.

3.2 HyPro Template Polyhedra Implementation

In this section we will focus on some observations made on template polyhedra and their operations that occurred during their implementation in the C++ library *HyPro* [SÁBMK17, Sch19]. Some operations need to be explored more in depth that have not been further considered in the previous chapter; for instance, how can an affine transformation be applied on some template polyhedron, an operation that is definitely required for the reachability analysis? We are going to look over the operations of checking emptiness, applying an affine transformation and intersection, we will explain some optimizations implemented and how they work. Additionally, we will be covering fixed point detection as an extra improvement.

Emptiness. Checking for emptiness is necessary for determining whether we intersected certain states S with our current set X or not, as $X \cap S \neq \emptyset \iff X$ intersects S. But checking for emptiness can also be used to shortcut other operations since they do not have to be exerted when at least one operand is empty. For example, $X \cup \emptyset = X$ and $X \cap \emptyset = \emptyset$. Therefore, it makes sense to compute the emptiness of a template polyhedron when it is demanded, and to cache the result. When emptiness is checked again, the cached result can be returned. Since a template polyhedron $\langle H,c \rangle$ is empty when $Hx \leq c$ is infeasible, emptiness can be checked via an LP call. This emptiness result can only change when the offset vector changes, in which case the emptiness can be recomputed using an LP call.

Affine Transformation. The main problem with affine transformations $x' = Ax + b$ for some matrix $A \in \mathbb{R}^{dxd}$ and $b \in \mathbb{R}^d$ on template polyhedra is that it is defined for vertices $x \in \mathbb{R}^d$. The naive approach for implementing affine transformation would then be to first, assuming that the template polyhedron is bounded, treat the template polyhedron as a H-polytope. As a second step, its vertices $\{v_0,...,v_k\}$ with $k \in \mathbb{N}$ can be enumerated and $v_i' = Av_i + b$ can be applied. The resulting vertices $\{v_0',...,v_k'\}$ can then be turned into intersecting halfspaces again, but the new halfspaces $\langle G,d \rangle$ do not necessarily have the same normal vectors as the halfspaces in $\langle H,c \rangle$. Since we want to enable the efficiency of operations between template polyhedra with the same template, we overapproximate the transformed set $\langle G,d \rangle$ by the template directions H. Only then the affine transformation is closed with respect to the representation under the template H.
Both the vertex enumeration and the facet enumeration are costly operations [F+04] and are to be avoided in general. The following proposed method accomplishes to avoid both enumerations. It is based on the observation that one can directly

overapproximate in the transformed directions $H \cdot A$ instead of transforming the template polyhedron $\langle H,c\rangle$ and overapproximate in the directions H afterwards. The support values of the overapproximation in direction HA are then shifted by the value $H \cdot b$ to ultimately overapproximate the transformed set in directions H. This approach is very similar to the affine transformation rule of the support functions.

Lemma 3.2.1. *Let $\langle H,c\rangle$ be a template polyhedron in $\mathbb{R}^d$. Let $f : \mathbb{R}^d \to \mathbb{R}^d, x \mapsto Ax+b$ be an affine transformation with $A \in \mathbb{R}^{d\times d}$ and $b \in \mathbb{R}^d$. Let $\alpha_L(S) \in \mathbb{R}^d$ be a vector where every i-th entry has the value $\max L_i x$ subject to $x \in S$ for $S \subseteq \mathbb{R}^d$ and $L \in \mathbb{R}^{n\times d}$. Then f applied on $\langle H,c\rangle$ is the template polyhedron $\langle H, \alpha_{HA}(\langle H,c\rangle) + H \cdot b\rangle$.*

Proof. After the transformation we are overapproximating. The overapproximation for each row H_i is obtained by computing the support value in direction H_i^T and thus is also obtainable by a support function. The support value of the transformed template polyhedron in direction H_i would be

$$\begin{aligned}\rho_{A\langle H,c\rangle+b}(H_i^T) &= \rho_{\langle H,c\rangle}(A^T H_i^T) + H_i^T b \\ &= \rho_{\langle H,c\rangle}((H_i A)^T) + H_i^T b\end{aligned}$$

If we look at the support values for every template row i as a vector in $\mathbb{R}^m$ we have

$$\begin{aligned}\begin{pmatrix} \rho_{A\langle H,c\rangle+b}(H_0^T) \\ \vdots \\ \rho_{A\langle H,c\rangle+b}(H_{m-1}^T)\end{pmatrix} &= \begin{pmatrix} \rho_{\langle H,c\rangle}((H_0 A)^T) + H_0^T b \\ \vdots \\ \rho_{\langle H,c\rangle}((H_{m-1} A)^T) + H_{m_1}^T b\end{pmatrix} \\ &= \begin{pmatrix} \max\ (H_0 A)^T x \text{ subj. to } x \in \langle H,c\rangle \\ \vdots \\ \max\ (H_{m-1} A)^T x \text{ subj. to } x \in \langle H,c\rangle\end{pmatrix} + \begin{pmatrix} H_0^T b \\ \vdots \\ H_{m-1}^T b\end{pmatrix} \\ &= \alpha_{HA}(\langle H,c\rangle) + Hb\end{aligned}$$

Since the vector of support values $\alpha_{HA}(\langle H,c\rangle) + Hb$ contains the support functions of the transformed template polyhedron but overapproximated into the template directions H, the transformed template polyhedron is $\langle H, \alpha_{HA}(\langle H,c\rangle) + Hb\rangle$. □

Note that the overapproximation in the affine transformation introduces approximation error. This approximation error again gets overapproximated in the next application of the affine transformation, thus the template polyhedra suffer from the same wrapping effect as boxes using the presented algorithm for affine transformation. A more sophisticated choice of the template can reduce the accumulated approximation error.

Example 3.2.1. *We will demonstrate an affine transformation based on Figure 3.3. Assume we are given:*

$$Ax + b = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2}\end{pmatrix}\begin{pmatrix} x \\ y\end{pmatrix} + \begin{pmatrix} 3.5 \\ 2\end{pmatrix} \text{ and } \langle H,c\rangle = \left\langle \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1\end{pmatrix}, \begin{pmatrix} 2 \\ 0 \\ 2 \\ 0\end{pmatrix}\right\rangle$$

The affine transformation $Ax+b$ rotates a given point by 45 degrees in counter-clockwise direction around the origin and then translates the rotated point by b. To carry out the

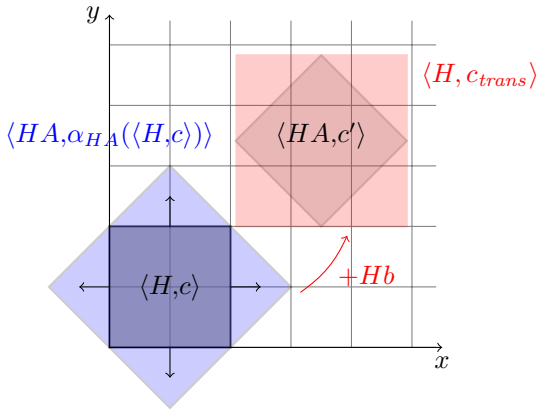


Figure 3.3: Affine Tranformation. Applying an affine transformation on $\langle H,c\rangle$ results in $\langle HA,c'\rangle$, which does not have the same templates H. To equalize both templates, first $\langle H,c\rangle$ is overapproximated by the directions of HA and this results in $\langle HA,\alpha_{HA}(\langle H,c\rangle)\rangle$. Then the transformed offset Hb is added to get $\langle H,c_{trans}\rangle$.

transformation according to the presented algorithm, the template directions need to be transformed first:

$$HA = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix}$$

Note that the directions HA are the same template directions as the one of the set we are trying to overapproximate. With the transformed directions HA we can now perform an overapproximation of $\langle H,c\rangle$ in the directions HA. Exemplary we will calculate the overapproximation with one direction.

$$\max \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \text{ subj. to } \begin{pmatrix} x \\ y \end{pmatrix} \in \langle H,c\rangle = \sqrt{2}$$
$$\Rightarrow \text{ Halfspace is: } \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \sqrt{2}$$

The optimal value is reached in point $\begin{pmatrix} 2 \\ 0 \end{pmatrix}$. *All other overapproximations are performed in a similar fashion. The overall approximation of* $\langle H,c\rangle$ *is:*

$$\langle HA, \alpha_{HA}(\langle H,c\rangle)\rangle = \begin{pmatrix} \frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ \frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \\ -\frac{\sqrt{2}}{2} & -\frac{\sqrt{2}}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \leq \begin{pmatrix} \sqrt{2} \\ \sqrt{2} \\ 2\sqrt{2} \\ 0 \end{pmatrix}$$

Which can be seen in Figure 3.3 as the blue diamond shape enclosing $\langle H,c\rangle$. *In the*

next step Hb needs to be computed:

$$Hb = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 3.5 \\ 2 \end{pmatrix} = \begin{pmatrix} 3.5 \\ -3.5 \\ 2 \\ -2 \end{pmatrix}$$

which we can now use Hb to acquire the overapproximation of $\langle HA,c' \rangle$ called $\langle H,c_{trans} \rangle$:

$$\langle H,c_{trans} \rangle = \langle H, \alpha_{\langle H,c \rangle}(HA) + Hb \rangle = \left\langle H, \begin{pmatrix} \sqrt{2}+3.5 \\ \sqrt{2}-3.5 \\ 2\sqrt{2}+2 \\ -2 \end{pmatrix} \right\rangle$$

which is the red overapproximation of $\langle HA,c' \rangle$ in Figure 3.3.

Intersection. The efficient intersection of two canonical template polyhedra $\langle H,c \rangle$, $\langle G,d \rangle$ as proposed only works if $H = G$. In reality, this is not always the case. If one knows that each row in G is also a row in H, then, with a naive pairwise comparison scheme, $\mathcal{O}(m^2)$ comparisons are needed to find every row of G in H.
In the case that $H \in \mathbb{R}^{m \times d}$ and $G \in \mathbb{R}^{n \times d}$ are different, one possible algorithm to compute the intersection would be to first overapproximate $\langle G,d \rangle$ with the template H and then to intersect them. The overapproximation has a running time of $\mathcal{O}(m \cdot LP)$, while the intersection can run in $\mathcal{O}(m)$, since now both the overapproximation and $\langle H,c \rangle$ use the same template H. This procedure has an overall complexity of $\mathcal{O}(m \cdot LP)$.
The implemented algorithm uses a combination of both algorithms: At first, we check for each row G_i in G whether G_i is a row in H. If this is the case, then we can mark G_i as found and resort to the efficient intersection and return the smaller offset value $e_i = \min\{d_i,c_j\}$ of the halfspaces $G_i x \leq d_i$ and $H_j x \leq c_j$ where $j \in [0,n-1]$. By doing so, we save an optimization query. In the case that G_i does not equal any row in H, we add $G_i x \leq d_i$ as a new constraint to another template polyhedron $\langle J,f \rangle$, which initially equals $\langle H,c \rangle$. If all rows in G were found then we can return the intersection $\langle H,e' \rangle$ where $e'^T = \begin{pmatrix} e'_0 & \dots & e'_{m-1} \end{pmatrix}$ with $e'_i = e_i$ if there was a row in G found for this row and $e'_i = c_i$ else. On the other hand, if not all rows have been found, then $\langle J,f \rangle$ will equal $\langle H,c \rangle$ extended with all constraints of G whose rows have not been found. We can then solve a LP $\max H_i x$ subj. to $x \in \langle J,f \rangle$ for each direction H_i that is not equivalent to some row in G to obtain the respective offset. This last step is a reduced overapproximation, where we do not optimize into the directions of H where the offset value of the overapproximation is already known by the previous step.
This method needs $\mathcal{O}(m^2)$ row comparisons in the ideal case that all rows of G are already within H. In any other case, it needs $\mathcal{O}(m^2 + m \cdot LP)$. So although this algorithm has a worse overall runtime complexity, under the right conditions it can be faster.

Example 3.2.2. *Let us go through an example for the intersection. For this we will compute the intersection of the template polyhedra displayed in Figure 3.4. It is:*

$$\langle H,c \rangle = \left\langle \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 3 \\ 0 \\ 2 \\ 2 \end{pmatrix} \right\rangle \text{ and } \langle G,d \rangle = \left\langle \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 4 \\ 2.5 \\ 6 \end{pmatrix} \right\rangle$$

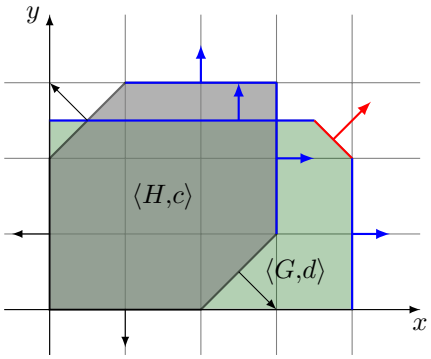


Figure 3.4: Intersection of two template polyhedra $\langle H,c\rangle$ with $\langle G,d\rangle$. The intersection of the blue halfspaces can carried out efficiently. Only for the red halfspace of $\langle G,d\rangle$ an LP needs to be called.

Through row-wise comparison from each row in G with each row in H we can infer that $G_0 = H_0 = \begin{pmatrix}1 & 0\end{pmatrix}$ and $G_1 = H_2 = \begin{pmatrix}0 & 1\end{pmatrix}$. For both rows, the intersection can be computed efficiently by taking the smaller offset as both point into the same direction: The result of the first pair is $H_0 x \leq 3$ and of the second pair is $H_2 x \leq 2.5$. Only for G_2 we cannot find a row in H that is equal. Thus we intersect by generating

$$\langle J,f\rangle = \left\langle \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \\ 1 & -1 \\ 1 & 1 \end{pmatrix}, \begin{pmatrix} 3 \\ 0 \\ 3 \\ 0 \\ 2 \\ 2 \\ 6 \end{pmatrix} \right\rangle$$

and computing $\max H_i x$ subj. to $x \in \langle J,f\rangle$ for every H_i that has no equal row in G, since for the other rows we already know the value of the overapproximation, which in this example would be H_1, H_3, H_4, H_5. This operation is a reduced version of the overapproximation. Since none of these directions are cut off by $G_2 x \leq 6$, the end result would be $\langle H, \begin{pmatrix}3 & 0 & 2.5 & 0 & 2 & 2\end{pmatrix}^T\rangle$.

Fixed point Detection. Fixed point detection is a topic of high interest for the reachability analysis of hybrid automata since the successful detection of a fixed point can avoid the computation of whole flowpipes and therefore save valuable time. In general, a fixed point has been found if our current segment $\langle H,c_n\rangle$ is a subset of some other previously computed segment $\langle H,c_k\rangle$, so if $\exists k \in \mathbb{N}.\langle H,c_n\rangle \subseteq \langle H,c_k\rangle$ for $k < n$ [ACH+95]. Checking this condition requires a check for the subset relation, or differently expressed, *containment* of a template polyhedron within another template polyhedra, which is an efficient operation that can be carried out in linear time for template polyhedron as described in lemma 3.1.2.
A naive procedure would compare every newly computed segment to every previously computed segment to check for a fixed point. This however is time-consuming. For affine hybrid automata, it is enough to check for all previously computed initial sets

of every flowpipe whether the current initial set is contained, as trajectories coming from different non-intersecting initial sets cannot overlap. If these trajectories could overlap, then at the intersection point of these trajectories nondeterminism would prevail, violating the determinism of the affine flow. By using this trick, we can reduce the number of subset checks needed from $\mathcal{O}(N^2)$ to $\mathcal{O}(\#init^2)$ where $\#init \leq N$ is equal to the number of initial states at the end of the reachability computation and N is the number of all computed segments of every flowpipe accumulated.

3.3 Taylor Approximation-based Reachability

Next we will inspect an alternate reachability analysis algorithm proposed by Sankaranarayanan et al. in [SDI08b]. It integrates nicely into the framework of flowpipe-based algorithms as it computes a first segment and applies time evolution to it, but does so differently. Its advantage is that vertex enumeration is not needed during the whole computation; these computations are replaced by multiple LP calls. The main idea for both the first segment computation as well as the time evolution is the same: To compute the first/next segment, approximate the trajectory of the linear function induced by a template constraint for every template constraint via Taylor approximation and maximize its value within the time segment to ensure overapproximation. As the trajectory of a template constraint is defined by the flow of the location, the change of the template constraint needs to be computed, which can be done via the Lie derivative. Both the Taylor approximation and the Lie derivative will be explained before the first segment computation and the time evolution will be explained.

Lie Derivative. The Lie derivative is a powerful tool in analysis and general relativity and is named after Sophus Lie, a norwegian mathematician. It is able capture the change of a tensor field (i.e. functions and vector fields) on a differentiable manifold (for instance $\mathbb{R}^d$ or some convex set therein) along another vector field [War13]. For this thesis, we will only consider its abilities in the context of functions in $\mathbb{R}^d$ along affine vector fields. It can be seen as the equivalent of the derivative but under the influence of an underlying vector field. Intuitively, it denotes the change of a function h while moving along a vector field D. As an informal example, imagine a boat on a river with a scientist in the boat (or look at Figure 3.5). While the boat floats along the stream of the river, the scientist looks checks the temperature of the air. The change of the temperature under the influence of the river is exactly what the Lie derivative describes. Hereby, the river represents the underlying vector field while the temperature of the air represents the function that is under the influence of the river.

Definition 3.3.1. *(Lie Derivative)*
Let $f_0(x),...,f_{d-1}(x)$ be differentiable functions in $\mathbb{R}^d$. The Lie derivative of a continuous and differentiable function h over a vector field $D(x) = \langle f_0(x),...,f_{d-1}(x)\rangle$ in $\mathbb{R}^d$ is defined as [War13]:

$$\mathcal{L}_D(h) = \nabla h \cdot D(x) = \sum_{i=0}^{d-1} \frac{\partial h}{\partial x_i} \cdot f_i(x)$$

Deriving the Lie derivative is out of the scope of this thesis, but one can interpret the formal definition in this way: All points of the partial derivatives of h are transformed according to the vector field D. We will use the Lie derivative to get the change of a template row H_i along the flow given by the location.

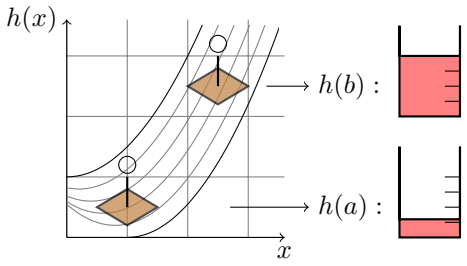


Figure 3.5: A scientist on a boat, who measures the temperature h at two different points a,b while floating along the river. The change of h under influence of the river is the Lie derivative.

Example 3.3.1. *Let $h(x,y) = 2x-4y+6$ be a function and $D(x,y) = \langle f_0(x,y), f_1(x,y)\rangle = \langle 2,1\rangle$ be an affine vector field. Since D defines the differential equations $\dot{x} = f_0(x,y) = 2$ and $\dot{y} = f_1(x,y) = 1$, D shifts x by two every time unit and y by one. The Lie derivative is then:*

$$\begin{aligned}\mathcal{L}_D(h(x,y)) &= \frac{\partial h(x,y)}{\partial x} f_0(x,y) + \frac{\partial h(x,y)}{\partial y} f_1(x,y) \\ &= 2 \cdot 2 - 4 \cdot 1 = 0\end{aligned}$$

We can see in the last line that the partial derivative in y-direction has not been changed by D. $\frac{\partial h(x,y)}{\partial x} = 2$ however has been displaced to $\frac{\partial h(x,y)}{\partial x} = 4$ according to the differential equation $\dot{x} = 2$.

Computing the Lie derivative reduces to the computation of d scalarproducts for linear multivariate functions $h(x_0,\ldots,x_{d-1}) = \sum_{j=0}^{d-1} a_j x_j + c$ where $a_j, c \in \mathbb{R}$. The calculation of a partial derivative for linear multivariate functions simplifies to a linear time coefficient search as all terms not containing x_i vanish:

$$\frac{\partial h(x_0,\ldots,x_{d-1})}{\partial x_i} = \frac{\partial \left(\sum_{j=0}^{d-1} a_j x_j + c \right)}{\partial x_i} = a_i$$

To obtain the Lie derivative, multiply the partial derivative vector with the respective vector field functions, resulting in d scalarproducts.

Taylor Approximation. The Taylor approximation, another useful tool from the field of mathematical analysis, is used to approximate a differentiable function of any shape by a polynomial in a region around a fixed development point [For84]. The quality of the approximation depends on the degree of the polynomial; the higher the degree the better the approximation. This quality increase in approximation is displayed in Figure 3.6. The Taylor approximation exactly describes the function when the polynomial has an infinite degree. Note that the Taylor approximation of a function can be under- as well as overapproximating.

Definition 3.3.2. *(Taylor Approximation) Let h be a continuous and differentiable function to at least the $m+1$-th degree. The Taylor approximation [For84] of h up to*

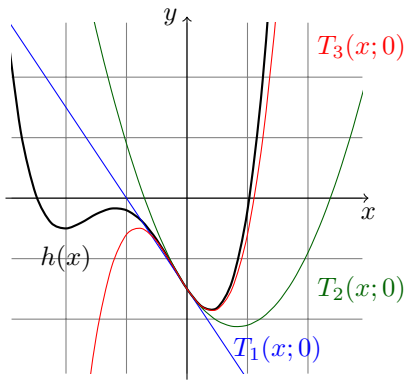


Figure 3.6: Taylor approximations of a function $h(x)$. Taylor approximation $h(x)$ at development point $x = 0$ of the first order in blue, of second order in green and of third order in red.

order m at development point a is defined as:

$$T_m h(x;a) = h(a) + h^{(1)}(a)(x-a) + ... + \frac{h^{(m)}(a)}{m!}(x-a)^m + \frac{h^{(m+1)}(\theta)}{(m+1)!}(x-a)^{m+1}$$

where $\theta \in [a,x+a]$ is an unknown scalar and $h^{(m)}(x)$ is the m-th derivative of h. The $m+1$-th term is known as the remainder.

First Segment. The goal of the first segment computation is the same as in Section 2.3: Given an initial set X_0 and the dynamics of a location L, compute an overapproximation $\hat{\mathcal{R}}_{[0,\delta]}$ of $\mathcal{R}_{[0,\delta]}$ that covers all reachable state within the time interval $[0,\delta]$. With our state set representations being the template polyhedra, the goal refines to: Given an initial set $\langle H,c_{init}\rangle$, compute an overapproximation $\langle H,c_0\rangle$ of $\mathcal{R}_{[0,\delta]}$. Let us introduce some notational remarks. For a template constraint $H_i = (h_{i,0} \dots h_{i,d-1})$ the scalarproduct $H_i x(t) = (h_{i,0} \dots h_{i,d-1})^T (x_0 \dots x_{d-1}) = \sum_{j=0}^{d-1} h_{i,j} x_j$ denotes a multivariate function. Also, with $H_i^{(j)} x(t)$ we denote the j-th order Lie derivative of the function $H_i x(t)$, for example, for some vector field D, we have $H_i^{(2)} x(t) = \mathcal{L}_D(\mathcal{L}_D(H_i x(t)))$. The following derivation of the technique can be found in [SDI08b]. For this method we assume that $\langle H,c_{init}\rangle$ and $\langle H,inv\rangle$ are non-empty and bounded. A method to use Taylor approximation-based reachability analysis with unbounded invariants can be found in Section 3.4.
The key idea is to approximate the trajectory of each template constraint H_i via Taylor approximation for the first time interval. Since the dynamics of the trajectory are dependent on the flow of the location, the Lie derivative is used instead of the standard derivative in every term of the Taylor approximation. The development point is $t = 0$, as we have the most information in form of $\langle H,c_{init}\rangle$ at this point. The goal is to get $H_i x(t)$. Therefore we want to approximate the trajectory of $H_i x(t)$ at $t = 0$.

If we take a look at the Taylor expansion

$$H_i x(t) = H_i x(0) + H_i^{(1)} x(0)(t-0) + ... + \frac{H_i^{(m)} x(0)}{m!}(t-0)^m + \frac{H_i^{(m+1)} x(\theta)}{(m+1)!}(t-0)^{m+1}$$
$$= H_i x(0) + H_i^{(1)} x(0)t + ... + \frac{H_i^{(m)} x(0)}{m!}t^m + \frac{H_i^{(m+1)} x(\theta)}{(m+1)!}t^{m+1}$$

then every $H_i^{(j)} x(0)$ for $j \in [1,m]$ is known, as it is just the j-th Lie derivative of the i-th initial constraint. The problem that occurs is that the Taylor approximation itself does not guarantee overapproximation, which is unhelpful; after all, we desire to definitely overapproximate the trajectory. An example of an underapproximation is demonstrated as the red graph in Figure 3.7. Hence, we seek to find an upper bound to the Taylor approximation. To achieve this, we can view the Taylor approximation as a function dependent on t, and every individual term therein can be maximized within the bounds of $x(0)$, which are given by the initial set $\langle H,c_{init}\rangle$. Therefore we can define for all $j \in [0,m]$:

$$a_{ij} := \max \frac{H_i^{(j)} x}{j!} \text{ subject to } x \in \langle H,c_{init}\rangle$$

and it holds $\frac{H_i^{(j)} x}{j!} \leq a_{ij}$. Another problem occurs when we consider the remainder term $\frac{H_i^{(m+1)} x(\theta)}{(m+1)!}$. Since it contains $x(\theta)$ with some unknown time value θ we cannot bound it by $\langle H,c_{init}\rangle$ as at time point θ, the valuation $x(\theta)$ is not guaranteed to satisfy $\langle H,c_{init}\rangle$. If we do not find an upper bound, then the maximization could possibly be unbounded. The remedy in this situation is the location invariant $\langle H,inv\rangle$: It bounds all constraints and sets that we compute while we are in the current location. Using the same technique as before, we can then define:

$$a_{i,m+1} := \max \frac{H_i^{(m+1)} x}{(m+1)!} \text{ subject to } x \in \langle H,inv\rangle$$

and it holds $\frac{H_i^{(m+1)} x}{(m+1)!} \leq a_{i,m+1}$. If we use both definitions combined we get:

$$\begin{aligned} H_i x(t) &= H_i x(0) + H_i^{(1)} x(0)t + ... + \frac{H_i^{(m)} x(0)}{m!}t^m + \frac{H_i^{(m+1)} x(\theta)}{(m+1)!}t^{m+1} \\ &\leq a_{i0} + a_{i1}t + ... + a_{im}t^m + a_{i,m+1}t^{m+1} \\ &:= p_i(t) \end{aligned}$$

$p_i(t)$ is a univariate polynomial only dependent on the time t and is an upper bound to $H_i x(t)$ as well as its Taylor approximation, as it can be seen in Figure 3.7 as the green plot. Considering that we want to bound $p_i(t)$ in time interval $[0,\delta]$ by a value, which will be the new offset value, we can again search for the maximum value of $p_i(t)$ within $[0,\delta]$ for it holds $p_i(t) \leq \max p_i(t)$ *subject to* $t \in [0,\delta]$. With $p_i(t)$ being an univariate polynomial the point in time t_{max} where the maximum $p_i(t_{max})$ can be found, can be analytically acquired by computing the derivative $p_i'(t)$ and evaluating $p_i(t)$ at all $t = t_r$ where t_r is a root of $p_i'(t)$, $t = 0$ and $t = \delta$. When t_{max} is found, then $p_i(t_{max})$ will be the new offset of the row H_i. This process is repeated for all

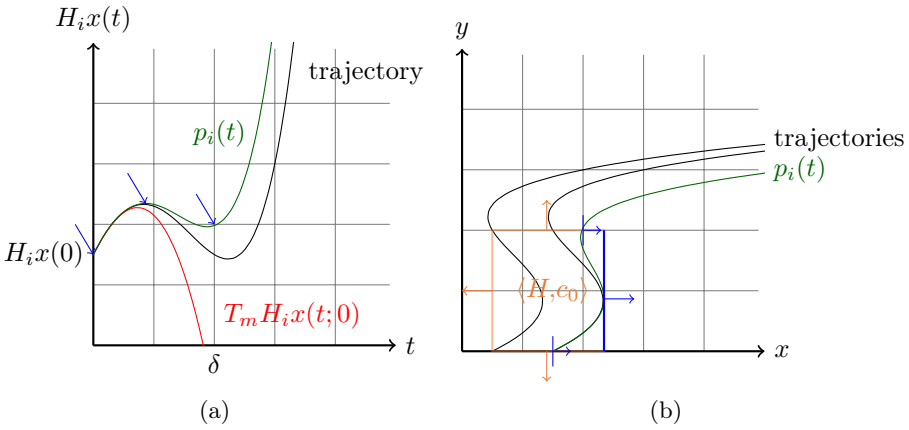
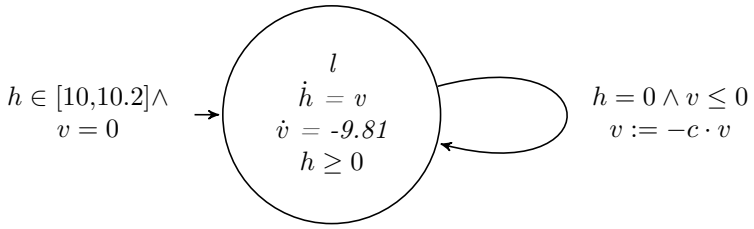


Figure 3.7: First Segment Computation. **(a)** In the plot of $H_i x(t)$ over time t, $T_m H_i x(t;0)$ underapproximates the actual trajectory $H_i x(t)$. $p_i(t)$ overapproximates the trajectory. The blue arrows mark potential maximal points at $t = 0$, $t = \delta$ and the root of $p_i'(t)$ in the interval $[0,\delta]$. **(b)** In this x-y-plot one can see the same trajectory, $p_i(t)$ and blue arrows. The big blue line marks the new halfspace $H_i x \leq c_{0,i}$. The first segment in orange $\langle H,c_0 \rangle$ emerges when this process is repeated for all template rows.

rows of the template H, see the right part of Figure 3.7.
To summarize, for every row $m+1$ LP calls are made to get an overapproximation to the Taylor polynomial modelling the trajectory of the current row. For the thereby assembled polynomial $p_i(t)$, the maximum can be found analytically, and that maximum is the new offset.

Example 3.3.2. *This example is based on the bouncing ball hybrid automaton from Example 2.2.1. This hybrid automaton models the height h and the velocity v of a ball that is dropped from some height between $[10,10.2]$ and then falls down. If the ball reaches the ground, it bounces back up. Let the time step size be $\delta = 0.1$. The automaton is depicted here:*



Let us assume our template is H is the same hexagonal template H from Example 3.1.1. Let us denote the variable vector by $x = \begin{pmatrix} h & v \end{pmatrix}^T$. The initial constraints

$h \in [10,10.2] \wedge v = 0$ *and invariant* $h \geq 0$ *expressed over* H *would be:*

$$\langle H,c_{init}\rangle = Hx \leq c_{init} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} h \\ v \end{pmatrix} \leq \begin{pmatrix} 10.2 \\ -10 \\ 0 \\ 0 \\ -10 \\ 10.2 \end{pmatrix} \text{ and } \langle H,inv\rangle = \langle H, \begin{pmatrix} \infty \\ 0 \\ \infty \\ \infty \\ \infty \\ \infty \end{pmatrix} \rangle$$

Since we are in the only location l*, we follow the flow defined by the vector field* $D(h,v) = \langle v, -9.81\rangle$*. Let us compute the offset of the first segment for the first template row* H_0x*. To start the computation, we need the first Lie derivatives:*

$$\begin{aligned} H_0x &= \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} h \\ v \end{pmatrix} = h \\ H_0^{(1)}x &= \mathcal{L}_D(H_0x) = \frac{\partial H_0x}{\partial h}v + \frac{\partial H_0x}{\partial v}(-9.81) = 1 \cdot v + 0 \cdot (-9.81) = v \\ H_0^{(2)}x &= \mathcal{L}_D(H_0^{(1)}x) = \frac{\partial H_0^{(1)}x}{\partial h}v + \frac{\partial H_0^{(1)}x}{\partial v}(-9.81) = 0 \cdot v + 1 \cdot (-9.81) = -9.81 \\ H_0^{(3)}x &= \mathcal{L}_D(H_0^{(2)}x) = 0 \end{aligned}$$

Interestingly, even if we would want to compute an approximation of higher order, the Lie derivatives of any order higher than two vanish. This leads to the conclusion that the approximation itself is exact. Now, following the algorithm, to get the overapproximation $p_i(t)$ *we need to compute the maximization queries* a_0j*:*

$$\begin{aligned} a_{00} &= \max H_0x \text{ subject to } x \in \langle H,c_{init}\rangle \\ &= \max h \text{ subject to } x \in \langle H,c_{init}\rangle && \text{//Reminder: } h \in [10,10.2] \\ &= 10.2 \\ a_{01} &= \max H_0^{(1)}x \text{ subject to } x \in \langle H,c_{init}\rangle \\ &= \max v \text{ subject to } x \in \langle H,c_{init}\rangle && \text{//Reminder: } v = 0 \\ &= 0 \\ a_{02} &= \max \frac{H_0^{(2)}}{2!}x \text{ subject to } x \in \langle H,c_{init}\rangle \\ &= \max \frac{-9.81}{2} \text{ subject to } x \in \langle H,c_{init}\rangle \\ &= -4.905 \end{aligned}$$

This results in $p_i(t) = 10.2 - 4.905t^2$*. The derivative is* $p_i'(t) = -9.81t$*. Since the root of* $p_i'(t)$ *is at* $t_{\max} = 0$ *which is in* $[0,0.1]$*, the new offset is* $p_i(t_{\max}) = p_i(0) = 10.2$*. This makes sense, as figuratively speaking the ball in the model can only fall down when starting from a height* $h \in [10,10.2]$*. So after one time step, the height of the ball is bounded from above by* 10.2*.*

Time Evolution. After the computation of the first segment, the goal is as follows: Given a segment $\langle H,c_k\rangle$ compute the next segment $\langle H,c_{k+1}\rangle$ that covers the trajectories in time segment $[k\delta,(k+1)\delta]$. The idea is the similar to the first segment computation: Taylor expansion is used to approximate the trajectory of every row

H_i. This time, $H_i x(t)$ at some time point t is given and $H_i x(t+\delta)$, the row after one time step, must be approximated. Therefore the development point t is chosen for the Taylor expansion:

$$\begin{aligned} H_i x(t+\delta) &= H_i x(t) + ... + \frac{H_i^{(m)} x(t)}{m!}(t+\delta-t)^m + \frac{H_i^{(m+1)} x(t+\theta)}{(m+1)!}(t+\delta-t)^{m+1} \\ &= H_i x(t) + H_i^{(1)} x(t)\delta + ... + \frac{H_i^{(m)} x(t)}{m!}\delta^m + \frac{H_i^{(m+1)} x(t+\theta)}{(m+1)!}\delta^{m+1} \end{aligned}$$

where $\theta \in [0,\delta]$ is some unknown value. Every term is either dependent on $x(t)$ or $x(t+\delta)$, thus we can exclude them and summarize the remaining terms:

$$\begin{aligned} H_i x(t+\delta) &= \left(H_i + H_i^{(1)}\delta + ... + \frac{H_i^{(m)}}{m!}\delta^m \right) \cdot x(t) + \left(\frac{H_i^{(m+1)}}{(m+1)!}\delta^{m+1} \right) \cdot x(t+\theta) \\ &=: g_i^T x(t) + r_i^T x(t+\theta) \end{aligned}$$

g_i is the sum of the first m Taylor expansion terms and can be geometrically interpreted as the direction with the highest degree of approximation of the trajectory, while r_i, the $m+1$-th term, can be read as the direction that would additionally improve the approximation, provided it is added at $t+\theta$. Both directions can be seen in Figure 3.8.
To find a suitable upper bound for $H_i x(t+\delta)$, we further inspect $x(t)$ and $x(t+\theta)$: Considering that $x(t)$ is the valuation at time t, we know that $x(t)$ must lie in the current segment $\langle H,c_k \rangle$, so $x(t) \in \langle H,c_k \rangle$. Furthermore, $x(t+\theta)$ can lie outside $\langle H,c_k \rangle$, since at time point $t+\theta$, the valuation could have left $\langle H,c_k \rangle$ due to the flow. The situation is similar to the first segment computation, where $x(\theta)$ also needed to be bounded. This was solved by bounding $x(\theta)$ by the location invariants; analogously, we can apply the same solution here: Since the location invariants bound every valuation that gets computed within the location, it must at least hold $x(t+\theta) \in \langle H,inv \rangle$. To ensure overapproximation of the next segment, we can define the following maximization queries:

$$\begin{aligned} g_i^{\max} &:= \max g_i^T x \text{ subject to } x \in \langle H,c_k \rangle \\ r_i^{\max} &:= \max r_i^T x \text{ subject to } x \in \langle H,inv \rangle \end{aligned}$$

Using these maximization queries we find an upper bound to $H_i x(t+\delta)$ since

$$H_i x(t+\delta) = g_i^T x(t) + r_i^T x(t+\theta) \leq g_i^{\max} + r_i^{\max}.$$

This bound is used as the new offset for H_i, see Figure 3.8. Overall, only the Lie derivatives and the two maximization queries need to be computed for each template row H_i to compute the next segment. Although this method works without vertex enumeration it has one major disadvantage: For the reason that the remainder term is bounded by the invariant, which can be a large overapproximation, this method can be unprecise. This flaw can be limited by the technique that is explained in the next Section 3.4, location invariant strengthening.

Example 3.3.3. *To continue example 3.3.2, we compute the time evolution of the first constraint $H_0 x \leq 10.2$ and the goal is to find the upper bound of $H_i x(2\delta)$. Recall*

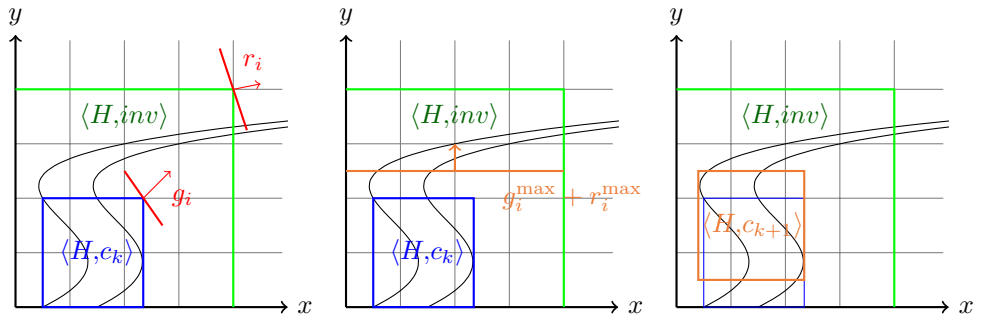


Figure 3.8: Time evolution. **Left:** Maximization directions g_i and r_i and their bounds $\langle H,c_k\rangle$ and $\langle H,inv\rangle$. **Middle:** $g_i^{\max} + r_i^{\max}$ is the new offset for the upper row. **Right:** This process is repeated for every row to obtain $\langle H,c_{k+1}\rangle$.

that $\delta = 0.1$. *Completing the first segment computation for all rows, we come to the conclusion that the first segment* $\langle H,c_1\rangle$ *consists of the constraints* $h \in [10,10.2] \wedge v = -0.981$. *The first three Lie derivatives have already been calculated in 3.3.2 and can now be summarized to get* g_i.

$$
\begin{aligned}
g_0 &= H_i + H_i^{(1)}\delta + \frac{H_i^{(2)}}{2!}\delta^2 \\
&= \begin{pmatrix}1\\0\\0\end{pmatrix} + \begin{pmatrix}0\\\delta\\0\end{pmatrix} + \begin{pmatrix}0\\0\\-4.905\delta^2\end{pmatrix} = \begin{pmatrix}1\\\delta\\-4.905\delta^2\end{pmatrix} = \begin{pmatrix}1\\0.1\\-0.04905\end{pmatrix}
\end{aligned}
$$

The maximization into direction g_i *gives:*

$$
\begin{aligned}
g_0^{\max} &= \max g_0^T x \text{ subject to } x \in \langle H,c_1\rangle \\
&= \max h + 0.1v - 0.04905 \text{ subject to } x \in \langle H,c_1\rangle \\
&= 10.2 + 0.1 \cdot (-0.981) - 0.04905 = 10.05285
\end{aligned}
$$

Since the Lie derivatives of order three and higher equal zero, we omit the remainder term r_0. *Therefore, the offset of the next segment for row* H_0 *is* $g_0^{\max}$. *This makes sense, insomuch as the ball has fallen down only a little from height 10.2 to 10.05285 during 0.1 time units.*

3.4 Location Invariant Strengthening

As seen in the previous section, both the first segment computation as well as the time evolution for template polyhedra in the Taylor approximation-based reachability algorithm depend on the location invariant $\langle H,inv\rangle$ for bounding the remainder term. In both instances, a non-tight invariant leads to inprecision during the computation, while a tight invariant can increase the precision. The proposed method of *location invariant strengthening* (LIS) described in [SDI08a] accomplishes that: Based on the idea of positive invariants, which are sets where on every point on their surface the vector field points inside the set, a policy iteration technique is conducted to iteratively acquire a strengthened invariant until it converges up to a fixed error tolerance. The

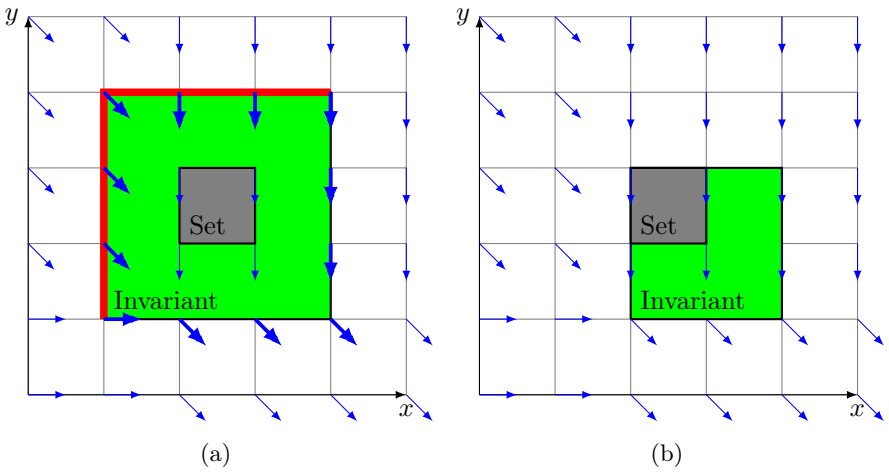


Figure 3.9: Location Invariant Strengthening Idea. **(a)** For positive invariance, at every point on the surface of the invariant it is checked whether the vector field points inside the invariant. The upper and left border are positive invariant, the right and lower border are not. **(b)** Location invariant strengthening truncates positive invariant borders iteratively to tighten the invariant constraints.

basic concept can be seen in Figure 3.9. Keep in mind that this method only works on bounded invariants and initial sets and does not guarantee to converge to the tightest possible invariant. At the end of the chapter however we present a method where LIS is used to shrink estimated bounds in models with unbounded invariants.

Positive Invariant. The key property that is checked during location invariant strengthening is positive invariance. Once a trajectory enters a positive invariant set, it can never leave it as the vector field pushes it back into the set. As already mentioned, if we look at a closed set S in a vector field, then S is positive invariant if at every point on its surface, the vector field points inside S [Bla99]. We can check whether the vector field D points inside a closed template polyhedron $\langle H,c\rangle$ by computing the Lie derivative $L_D(H_i x)$ for each row i and checking whether $L_D(H_i x) > 0$. If $L_D(H_i x)$ is greater than zero, then we know that on the hyperplane $H_i x = c_i$ the trajectory of $H_i x$ would follow the flow. As we want to check this property only on the surface of $\langle H,c\rangle$ and not on the whole hyperplane, the check must at the same time satisfy $Hx \leq c$. We will define positive invariance in the context of all possible closed invariants that are between the initial template polyhedron $\langle H,c_{init}\rangle$ and the original location invariant $\langle H,inv\rangle$ as those are the strengthened invariants we are interested in.

Definition 3.4.1. *(Positive Invariant Set)*
Let $\langle H,c_{init}\rangle$ be the initial set and $\langle H,inv\rangle$ the location invariant.
For $c_{init} \leq c \leq inv$ the template polyhedron $\langle H,c\rangle$ is positive invariant [SDI08a] with respect to $\langle H,inv\rangle$ if and only if

$$\forall i \in [0,m-1] : (c_i = inv_i) \vee ((Hx \leq c \wedge H_i x = c_i) \models L_D(H_i x) > 0)$$

We can check positive invariance by solving the following linear program for each

row where $d_i < inv_i$,

$$\max L_D(H_i x) \text{ subj. to } x \in \langle H,c\rangle \wedge H_i x = c_i$$

and then checking whether the solution is greater equal 0. The condition that needs to be checked for positive invariance is hard to compute using template polyhedra due to the equation $H_i x = c_i$. To ease this problem, *Lagrangian relaxation* has been used by Sankaranarayanan et al. [SDI08a] to put more difficult constraints into the objective function.

Definition 3.4.2. *(Lagrangian Relaxation)*
Let $A \in \mathbb{R}^{m\times d}, B \in \mathbb{R}^{n\times d}$ and $a \in \mathbb{R}^m$, $b \in \mathbb{R}^m$, $c,x \in \mathbb{R}^d$. Let

$$P : \max c^T x \text{ subject to } Ax \leq a \wedge Bx = b$$

be a linear programming problem. Then

$$P_{rel} : \max c^T x + \mu^T (Bx - b) \text{ subject to } Ax \leq a$$

with non-negative weights $\mu \in \mathbb{R}^n_+$ is the Lagrangian relaxation of P.

The idea of the Lagrangian relaxation is to penalize the objective function if the equations $Bx = b$ get violated. The more they are violated, the higher the penalty. In this case, we will move the equation $H_i x = c_i$ into the objective function since this equation prevents us from using template polyhedra effectively. Note that the optimal solution to a relaxed maximization query P_{rel} is an upper bound to the optimal solution of the original problem P [BV04]. The consequences of relaxing the positive invariance are the *relaxed invariants*:

Definition 3.4.3. *(Relaxed Invariant)*
Let $\langle H,c_{init}\rangle$ be the initial set and $\langle H,inv\rangle$ the location invariant.
For $c_{init} \leq c \leq inv$ the template polyhedron $\langle H,c\rangle$ is a relaxed invariant [SDI08a] if and only if

$$\forall i \in [0,m-1] \text{ if } c_i < inv_i \text{ then } \langle H,c\rangle \models \mathcal{L}_D(H_i x - c_i) + \mu(H_i x - c_i) \leq 0$$

where $\mu \in \mathbb{R}_+$ is some arbitrary non-negative scaling factor.

Every relaxed invariant is a positive invariant [SDI08a]. To check relaxed invariance, one can simply make the maximization query

$$L_i : \max \mathcal{L}_D(H_i x - c_i) + \mu(H_i x - c_i) \text{ subj. to } x \in \langle H,c\rangle$$

and check whether the solution is smaller or equal to zero, which can be carried out as a maximization call on the template polyhedron $\langle H,c\rangle$. Since we assume $\langle H,c\rangle$ to be bounded, there is always a bounded optimum for L_i.

Policy Iteration. The overall structure of LIS is based on the structure of policy iteration, which is a technique not only used in reinforcement learning [Ber11], but also in static analysis [GGTZ07]. Policy iteration works in two steps: In the first step called policy evaluation, the current policies are examined and their optimality is measured. In the second step called the policy improvement, we take the results from the policy evaluation and derive new improved policies from them. This whole process

is repeated until the policies change no more, so a fixed point has been reached.
In the context of location invariant strengthening, our policies are the linear programs that verify which rows are relaxed invariants. Starting with $\alpha(0) = inv$, which is a relaxed invariant by definition, in the policy evaluation we will get the verification for each row i whether it is positive invariant. After having done that for all rows, we construct another linear program which delivers us strengthened invariant offsets $\alpha(j+1)$ for which it holds $\alpha(j) \geq \alpha(j+1)$ for $j \in [0,N]$ for some $N \in \mathbb{N}$. We continue this process until $\alpha(j) = \alpha(j+1)$. Thus, a sequence $\alpha(0) > ... > \alpha(N) = \alpha(N+1)$ of improved invariant offsets is generated, until a fixed point has been reached.

Policy Evaluation. Assume we are computing the $j+1$-th relaxed invariant and are thus in the j-th iteration of the policy iteration. During the policy evaluation step the goal is to obtain a verification, a certificate, that our current row i is a relaxed invariant. This is achievable via the *dual problem* of the so called *primal problem* since the optimal solution of the dual problem provides an upper bound to the optimal solution of the primal problem if the primal problem is a maximization [BV04]. In the dual problem, every constraint of L_i is turned into a variable; every variable of L_j is turned into a constraint and the direction of the objective function is inversed [BV04].

Definition 3.4.4. *(Dual Linear Program) Let $A \in \mathbb{R}^{m \times d}$, $b \in \mathbb{R}^m$ and $c,x \in \mathbb{R}^d$. Let $\max c^T x$ subject to $Ax \leq b$ be a linear program called the primal problem. Then the dual problem is defined as*

$$\min b^T \lambda \text{ subject to } A^T \lambda = c \wedge \lambda \geq 0$$

for the dual variable vector $\lambda \in \mathbb{R}^m$.

To simplify the notation, we omit the iteration index j from $\alpha(j)$ and just write α instead. The primal problem L_i for row i within the policy iteration scheme is:

$$L_i : \max \mathcal{L}_D(H_i x - \alpha_i) + \mu(H_i x - \alpha_i) \text{ subj. to } x \in \langle H,\alpha \rangle$$

where α_i denotes the i-th entry in the vector α. The result of the Lie derivative $\mathcal{L}_D(H_i x - \alpha_i)$ can be decomposed into a part containing variables $H_i' x$ and a variable-free offset $h_i \in \mathbb{R}$ such that $\mathcal{L}_D(H_i x - \alpha_i) = H_i' x + h_i$. Additionally, $\mu(H_i x - \alpha_i) = \mu H_i x - \mu \alpha_i$. $H_i' x$ and $\mu H_i x$ can be condensed into $(H_i' + \mu H_i)x$. All these transformations result in:

$$L_i : \max (H_i' + \mu H_i)x - \mu \alpha_i + h_i \text{ subject to } Hx \leq \alpha$$

Applying the dual transformation on L_i gives:

$$D_i : \min \alpha^T \lambda - \mu \alpha_i + h_i \text{ subject to } H^T \lambda = (H_i' + \mu H_i)^T \wedge \lambda \geq 0$$

If the solution of $D_i \leq 0$, then the optimal value of D_i certifies that the i-th row is indeed relaxed and this row is called *non-frozen*. The optimal point is saved for policy improvement. For a non-frozen row the offset can still be improved during policy improvement. If the solution of $D_i > 0$, then this row is termed a *frozen* row. The offset of a frozen row must not change, since there is an invariant that refrains it from changing. Therefore, the optimal point does not need to be saved for frozen rows; instead the row index is stored in a set of frozen row indices F for policy improvement. Both the row indices in the case of a frozen row as well as the optimal point in the

case of a non-frozen row are called *vertex certificates*. The vertex certificates are saved into a data structure for the policy improvement step.

Policy Improvement. The key datastructure for policy improvement is the *invariant certificate*. Essentially, it contains all vertex certificates of the non-frozen rows. From it, constraints can be generated which must hold for the next strengthened invariant. By minimizing under these constraints, we can obtain the biggest change to the next invariant.

Definition 3.4.5. *(Invariant certificate)*
An invariant certificate [SDI08a] is a tuple $\pi = (F,\Lambda)$ *with* $F \subseteq \{0,...,m-1\}$ *and* $\Lambda \in \mathbb{R}_+^{m\times m}$ *such that it holds:*

- *(frozen row) If* $i \in F$ *then the row* $\Lambda_i = 0$ *or*
- *(non-frozen row) if* $i \in [0,m-1] \setminus F$ *then* $H^T\Lambda_i = (H'_i + \mu H_i)^T \wedge \Lambda_i \geq 0$

The invariant certificate $\pi = (F,\Lambda)$ is filled during the policy evaluation phase. It validates for an invariant $\langle H,\alpha\rangle$ which constraints are exhausted and which invariants can be strengthened. All frozen rows $i \in F$ induce that $\alpha_i = inv_i$ as frozen rows are limited by inv_i, while for non-frozen rows it holds $\alpha^T\Lambda_i - \mu\alpha_i + h_i \leq 0$ (the objective function of D_i). Using invariant certificate π, we can derive constraints that must hold during policy improvement:

$$L_\pi : c_{init} \leq y \leq inv \wedge \bigwedge_{i\in F} y_i = inv_i \wedge \bigwedge_{i\in[0,m-1]\setminus F} \Lambda_i^T y - \mu y_i + h_i \leq 0$$

where $y \in \mathbb{R}^d$ is the new invariant offset vector. To get the smallest offset vector, we want to minimize each coefficient of y, with which we get to the linear program:

$$min \sum_{i=0}^{d-1} y_i \text{ subject to } L_\pi$$

The result of this LP call is the improved invariant vector of the current iteration $\alpha(j)$.

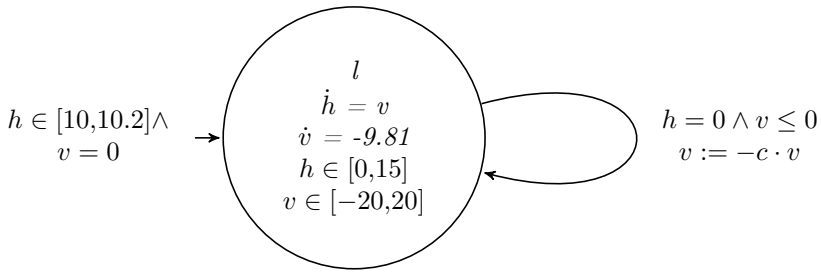
Location Invariant Strengthening. The whole algorithm for LIS is summarized in Algorithm 3.10. The complete procedure can be applied once before the start of every flowpipe computation when the initial set after the reset is known. In every iteration, the policy evaluation step solves m LPs (each to solve their respective D_i), then another LP call is made in the policy improvement step to solve L_{π_j}, so overall $m+1$ LP calls need to be made. Since the number of iterations is not fixed, the running time potentially rises to $\mathcal{O}(k(m+1)LP)$ for $k \in \mathbb{N}$. To definitely bound the number of iterations, LIS is stopped after $k' \in \mathbb{N}$ iterations in the implementation and the best approximation $\alpha(k')$ up to this point is returned.

Example 3.4.1. *Let us reconsider the example of the bouncing ball from example 3.3.2 again. This time, we modify the invariants to be bounded by constraints far away from the sets of reachable states: Let the invariants be* $h \in [0,15] \wedge v \in [-20,20]$. *The automaton then looks like this:*

Input: Initial set $\langle H, c_{init}\rangle$ Invariant $\langle H, inv\rangle$
Output: Strengthened invariant $\langle H, inv_{str}\rangle$

$\alpha(0) := inv$
repeat
  $\pi_j := \{\}$
  for each template row i in H **do**
    $\pi_j(i) :=$ Solve D_i
  end for
  Construct constraints L_{π_j} from π_j
  $\alpha(j) := \alpha(j+1)$
  $\alpha(j+1) := \min \sum_j y_j$ subj. to L_{π_j}
until $\alpha(j) = \alpha(j+1)$
return $\langle H, \alpha(j+1)\rangle$

Figure 3.10: Location invariant strengthening algorithm.



Let the template H also be only the first four rows of H from the example 3.3.2 to shorten this example. The constraints therefore represent a box. The initial constraints and the invariants would then be

$$\langle H,c_{init}\rangle = Hx \leq c_{init} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} h \\ v \end{pmatrix} \leq \begin{pmatrix} 10.2 \\ -10 \\ 0 \\ 0 \end{pmatrix} \text{ and } \langle H,inv\rangle = \langle H, \begin{pmatrix} 15 \\ 0 \\ 20 \\ 20 \end{pmatrix}\rangle$$

Let us compute the strengthened invariants given the initial constraints $h \in [10,10.2] \wedge v = 0$. We know from example 3.3.2 that the ball cannot rise but only fall, so we expect h to decrease (to zero) and also v to decrease as the speed decreases with every time step. Thus, the only positive invariant rows are the constraints $h \leq 15$ and $v \leq 20$; both constraints can be strengthened. $-20 \leq v$ cannot be strengthened because v develops towards negative infinity and it is not known up to which value.
Let us start with the policy evaluation step. We need to calculate

$$D_0 : \min \alpha^T \lambda - \mu\alpha_0 + h_0 \text{ subject to } H^T\lambda = (H_0' + \mu H_0)^T \wedge \lambda \geq 0$$

We choose $\mu = 3$. $H_0 = \begin{pmatrix} 1 & 0 \end{pmatrix}$ is known. The Lie derivative $\mathcal{L}_D(H_o x) = v$ is known from Example 3.3.2. Hence, $H_0' = \begin{pmatrix} 0 & 1 \end{pmatrix}$ and $h_0 = 0$ as $\mathcal{L}_D(H_0 x)$ did not have an variable-free offset other than zero. In the first iteration $\alpha^T = inv^T =$

$\begin{pmatrix}15 & 0 & 20 & 20\end{pmatrix}$. *These values simplify* D_0 *to:*

$$D_0 : \min \begin{pmatrix}15\\0\\20\\20\end{pmatrix}^T \begin{pmatrix}\lambda_0\\\lambda_1\\\lambda_2\\\lambda_3\end{pmatrix} - 45 \text{ subj. to } \begin{pmatrix}1 & -1 & 0 & 0\\0 & 0 & 1 & -1\end{pmatrix} \begin{pmatrix}\lambda_0\\\lambda_1\\\lambda_2\\\lambda_3\end{pmatrix} = \begin{pmatrix}3\\1\end{pmatrix} \wedge \lambda \geq 0$$

The minimization result of D_0 *returns 20 with* $\lambda_{D_0}^T = \begin{pmatrix}3 & 0 & 1 & 0\end{pmatrix}$. *Since* $20 > 0$, *the* 0*-th row is deemed a frozen row. Therefore, the value of this row will not be changed in this iteration.*
Solving D_i *for the other three rows it emerges that only the third row (with index two) is relaxed with* $h_2 = -9.81$, $\lambda_{D_2}^T = \begin{pmatrix}0 & 0 & 3 & 0\end{pmatrix}$ *and the optimal value -9.81, so row two is the only non-frozen row. This in return means only the upper bound* $v \leq 20$ *will be changed in this iteration. The invariant certificate after evaluating all four rows is* $\pi = (\{0,1,3\},\lambda_{D_2})$.
Now during policy improvement we can deduce following constraints from π*:*

$$L_\pi : \begin{pmatrix}10.2\\-10\\0\\0\end{pmatrix} \leq y \leq \begin{pmatrix}15\\0\\20\\20\end{pmatrix} \wedge \begin{pmatrix}0\\0\\3\\0\end{pmatrix}^T \begin{pmatrix}y_0\\y_1\\y_2\\y_3\end{pmatrix} - \mu y_2 + h_2 \wedge y_0 = 15 \wedge y_1 = 0 \wedge y_3 = 20$$
$$= \begin{pmatrix}10.2\\-10\\0\\0\end{pmatrix} \leq y \leq \begin{pmatrix}15\\0\\20\\20\end{pmatrix} \wedge y_0 = 15 \wedge y_1 = 0 \wedge y_3 = 20$$

As we can see, y_2 *is only bounded via* $0 \leq y_2 \leq 20$. *Consequently, in the minimization* $min \sum_{i=0}^{d-1} y_i$ *subject to* L_π *we can choose the lowest value within these bounds for* y_2, *so ultimately we get* $\alpha^T = \begin{pmatrix}15 & 0 & 0 & 20\end{pmatrix}$ *as the new invariant bounds. As expected, the former constraint* $v \leq 20$ *has been strengthened to* $v \leq 0$, *but we have also seen that not all constraints are immediately strengthened in the first iteration. Both the policy evaluation as well as the policy improvement will be repeated until there are no more changes in the invariant bounds.*

Unboundedness. Strictly speaking the Taylor approximation-based method and location invariant strengthening both only operate under the assumption of bounded invariants. If the invariants were unbounded, then at multiple points the techniques fail to provide useful information. For instance during the time evolution, if the maximization query of the remainder term $r_i^{\max} = \max r_i^T x$ subj. to $x \in \langle H,inv\rangle = \infty$ due to the unboundedness of $\langle H,inv\rangle$, then $H_i x(t)$ is bounded by $g_i^{\max} + r_i^{\max} = \infty$, from which we cannot generate a useful next segment.
In order to increase the range of models in which these techniques are usable, we automatically artificially bound the invariants with numbers that are not reachable within the time bound. Afterwards LIS can be used to strengthen the invariants to fit the current flowpipe computation. Currently the estimation of the numbers is set to a fixed bound 10^5 which is based on empiric knowledge about the benchmarks the techniques were tested on. Using the same technique one can manually bound all invariants in each location with a tighter bound depending on empiric knowledge. In this case the Taylor approximation-based method can work without LIS. A procedure

to automate the choice of the lowest unreachable bound is left for future work and is described in Chapter 5.
The policy iteration in LIS is known converge to saddle points [GSBS19]. Since the initial estimated invariant bounds leave room for unwanted saddle points, we keep track of the strengthened invariant with the smallest sum of coefficients $c_{smallest}$ during the policy iteration. Through the policy iteration $c_{smallest}$ is guaranteed to safely overapproximate the invariant region containing the flowpipe. By saving the invariant with the smallest sum of coefficient we take the smallest overapproximating strengthened invariant that occurred during the policy iteration. When the policy iteration in the j-th iteration converges to an invariant c, which could be a unwanted saddle point, we check whether $\sum_{i=0}^{d-1} c_i < \sum_{i=0}^{d-1} c_{smallest,i}$ and if that is the case then c is returned as the next strengthened invariant, else $c_{smallest}$ is returned.
Note that this method is not guaranteed to avoid all unwanted saddle points as the choice of $c_{smallest}$ could lead into another unwanted saddle point and is based on the assumption that LIS at one point generates a strengthened invariant that is coefficient-wise smaller than offsets of the saddle point we desire to avoid. A method that guarantees the evasion of all unwanted saddle points remains a topic of further research.

Cycle Detection. In practice we found out that the policy iteration can get stuck in a cycle as it revolves around different saddle points. These cycles often lead to the maximum number of iterations being reached, which were implemented to definitely bound the running time of LIS, which ultimately increased running time unnecessarily. In order to counter this phenomenon, we implemented a cycle detection where the strengthened invariant of the current iteration $\alpha(j)$ is compared with every strengthened invariants of all iterations. If there exists $k < j$ such that $\alpha(j) = \alpha(k)$, then we encountered a cycle and stop LIS, else we continue with the policy iteration. Since the number of iterations is bounded by some number $N \in \mathbb{N}$ for bounding the running time, the maximum amount of vector comparisons made by the cycle detection is bounded by $\mathcal{O}(N^2)$.

Numerical Corrections. While conducting experiments it became apparent that some LP calls returned infeasibility, although D_i and L_π are guaranteed to be feasible as $\langle H,c_{init}\rangle$ and $\langle H,inv\rangle$ are assumed to be non-empty and bounded. Further investigation revealed that numerical issues led to confusion about the feasibility. For instance, only if the solution of $D_i \leq 0$, then a row is deemed non-frozen, but a value could be $1.414 \cdot 10^{-16}$ instead, which is greater zero and therefore the row is deemed frozen. For this reason, *numerical corrections* were introduced: If the i-th row is deemed frozen but does not satisfy $\alpha_i >= inv_i$, then this row is only deemed frozen by numerical instabilities and the vertex certificate of this row is changed to the optimal solution of D_i. We refer to this correction as the *inserting correction*. On the other hand if the i-th row is deemed non-frozen but does not satisfy $\alpha^T \Lambda_i - \mu\alpha_i + h_i \leq 0$, then this row is falsely deemed non-frozen and the vertex certificate will be corrected to the index of the row i. This correction is labelled *removing correction*. The impact of these measures can be read in the next chapter under Section 4.5.

Chapter 4

Experimental Results

Previously, the theory of template polyhedra and their properties regarding the usage as state set representations during reachability analysis has been presented. Two different approaches have been proposed: The reachability analysis for affine hybrid automata following the algorithm described in Chapter 2.3 and the one introduced by Sankaranarayanan et al. in [SDI08b]. Both algorithms have been implemented in the library *HyPro* [SÁBMK17] and tested with the tool *HyDRA* [SÁ18]. In this chapter, we will refer to the template polyhedra using the HyPro state set representation interface the HyPro template polyhedra (short HTP), while the template polyhedra using the Taylor approximation-based reachability analysis Taylor template polyhedra (short TTP). To assess and compare the quality of both methods, they have been tested on several benchmarks and their results with special focus on running time and precision have been tabularized. This chapter includes an explanation of the experimental setup in the beginning and then continues to discuss the results in detail: The results for the general approach and for the Sankaranarayanan method and how they compare to each other are denoted in the Sections 4.2 and 4.3. In Section 4.4 the combination of support functions and template polyhedra are evaluated against both approaches. Afterwards, the benchmark results for location invariant strengthening and the fixed point detection are presented in Section 4.5.

4.1 Experimental Setup

HyPro and HyDRA. As mentioned before, both approaches have been implemented in the C++ library HyPro. HyPro offers the reachability analysis for affine hybrid automata as described in Section 2.3 with different state set representations and features a general interface for additionally implementing own state set representations [SÁBMK17]. It uses GLPK as a backend LP solver [Mak00], but also offers the possibility to integrate other LP solvers such as SMT-RAT [CKJ+15] in conjunction with exact arithmetic. The tool HyDRA is based on HyPro and extends its usability with several concepts, namely partial path refinement, parallelization and subspace decomposition [SÁ18]. During all the experiments, neither partial path refinement nor subspace decomposition were used. All computations are made with one thread, GLPK as the LP solver and inexact arithmetic.

Template Shape. As the choice of the template is crucial for the speed and precision of the template polyhedra, different compositions of the templates were tested. It makes sense to test their abilities with the same constraints as every other halfspace-based state set representation, so only with the initial constraints. Additionally, since the intersection operation is most efficient when the constraints of the invariants / guards / bad states are part of the template, we can also add these constraints gradually, to test their involvement in the running times. After adding all these constraints, it can still be possible that the template is rather unflexible and introduces a lot of approximation error when the constraints form a box. For this reason, also octagonal constraints have been tested.
Let $\langle H,c\rangle$ be the final template polyhedron. Let $\langle H_{init}, c_{init}\rangle$ be the template polyhedron representing the given initial constraints, $\langle H_{inv}, c_{inv}\rangle$ the invariants of the current location l, $\langle H_{guard}, c_{guard}\rangle$ the cumulated guards of all outgoing transitions from l and $\langle H_{bad}, c_{bad}\rangle$ the bad states. For two matrices $A \in \mathbb{R}^{m\times d}$ and $B \in \mathbb{R}^{n\times d}$ we define $A \mid B = \begin{pmatrix} A & B \end{pmatrix}^T$. Overall the proposed test templates are:

- *Only Init (OI)*: $H = H_{init}$ and $c = c_{init}$.
- *Initial and Invariants (II)*: $H = H_{init} \mid H_{inv}$ and $c = c_{init} \mid c_{inv}$.
- *Initial, invariants and guards (IIG)*: $H = H_{init} \mid H_{inv} \mid H_{guard}$ and $c = c_{init} \mid c_{inv} \mid c_{guard}$.
- *Initial, invariants, guards and bad states (IIGB)*: $H = H_{init} \mid H_{inv} \mid H_{guard} \mid H_{bad}$ and $c = c_{init} \mid c_{inv} \mid c_{guard} \mid c_{bad}$.
- *Octagon (OCT)*: All possible constraints where $x_i = \pm 1$ and $x_j \in \{-1,0,1\}$ for $i \neq j$ and $x_k = 0$ for $x_k \neq x_i$ and $x_k \neq x_j$.

Benchmarks. Next the benchmarks used for the experiments will be presented. All benchmarks were chosen for different challenges or because they have different difficulty levels within their challenge. The benchmarks are tabularized in Figure 4.1.

Bouncing Ball [CSM+ 15]. The bouncing ball (BB) is the easiest of all benchmarks as it is a two-dimensional system with one location and one transition. It has already been used in multiple examples throughout this theses, for instance Example 2.2.1, 3.3.2, 3.3.3 and as a bounded version in Section 3.4.1. A more detailed description of this system can be found at the beginning of Chapter 2.2. In this scenario, we assume the bad states to be $h \geq 10.3$. Since the bouncing ball benchmark is easy to solve, it is expected that it can be verified with all settings.

Building [TNJ16, CVD02]. The building benchmark describes the Los Angeles University Hospital which has multiple floors each having three degrees of freedom: Displacement in x- and y-direction and rotation. The bad states are reached when the displacement of in x-direction gets too high, so when $x_{25} >= 0.0051$. Overall, this benchmark has 50 dimensions and is chosen not only for its high dimensionality, but also for the high amount of precision needed to verify it. It is the benchmark with the highest dimension, but it has no transitions and lacks an invariant, therefore it is a purely continuous system.

Navigation [FI04]. The navigation benchmarks simulate point masses in the 2D plane with several regions with different dynamics. Each benchmark instance is an $n \times n$-grid. Every cell in this grid is its own location in the hybrid automaton and has a different flow dependent on the benchmark instance as well as invariants that mark the boundaries of the cell. Additionally, one cell is determined to be the destination the point masses should reach, aka the good states, and one cell is selected to be the bad states. For both the desired and the bad cell there is no flow defined. Every change from one cell to another is modelled via a transition. All instances are expected to not reach the bad states. The navigation benchmarks model the x- and y-coordinate as well as the velocities in x- and y- direction of the point mass and are therefore always a four dimensional system. This benchmark set has been chosen for its high branching factor as every cell in the grid has between two and four neighbouring cells it can transition to. Of the three navigation instances chosen, *navigation03 (NAV03)* is the easiest, while *navigation04 (NAV04)* visits more locations than navigation03 until it reaches the destination cell. *navigation09 (NAV09)* however increases the grid size from 3×3 to 4×4.

Platoon [MMH+11]. In the platoon benchmark we have three autonomously driven vehicles that drive behind each other. With every vehicle accelerating and decelerating differently, the safety property that needs to be checked is whether it can come to a collision within the platoon. For this reason, a minimum distance is defined that needs to be kept between every vehicle. Different instances of the platoon benchmark define different bounds on how much this minimum distance is allowed to be violated before the distance between two vehicles is counted as a collision. *Platoon42* leaves more room for violation while *Platoon30* has tighter bounds. The two locations of each benchmark model two phases in communication between the vehicles; in one location the vehicles can communicate about the distance to each other and adjust their speed accordingly, while in the second location the communication fails for a fixed amount of time, thus leading to uncoordinated de- and acceleration. The switch in communication is modelled by transitions that are enabled every five time units or higher. Although the benchmarks offer a high maximum number of jumps with 1000, the benchmarks are only able to jumps at most five times within their time bound of 20. Both instances are expected to be safe. The platoon benchmark has been chosen since it has a medium difficulty; precisionwise it needs less precision than the building benchmark and is also less dimensional with twelve variables, but in return it offers two locations and two transitions.

Space Rendezvous [CM17]. A *space rendezvous (SR01)* is known as the act of a spacecraft, the chaser, to attach itself to an orbiting body, the target, in space. In order to do that, the chaser first approaches the target which is modelled by a location. If it is close enough, it attempts the rendezvous, this is the second location. If during the approach or the rendezvous the chaser's velocity is too high or it misses the target, the rendezvous attempt failed, which are the bad states that need to be avoided. The rendezvous can only be attempted once, therefore a maximum of two jumps can be executed by this model, although it allows for infinitely many. This benchmark has been chosen for its needed precision, its many different invariant and bad state constraints and the high time bound of 300 time units.

Benchmark	Dim	Loc	Trans	Time	Max Jumps
BouncingBall	2	1	1	3	5
Building	50	1	0	20	0
Navigation03	4	9	24	1	6
Navigation04	4	9	24	3	6
Navigation09	4	16	48	3	6
Platoon30	12	2	2	20	1000
Platoon42	12	2	2	20	1000
SpaceRendezvous	5	3	3	300	∞

Figure 4.1: Benchmarks. The benchmarks and their dimension (Dim), their number of locations (Loc), number of transitions (Trans), maximum time bound per flowpipe (Time) and maximum number of jumps (Max Jumps) that can be taken.

Conditions. All experiments were conducted on a Intel core i7-7700HQ at 2.8 GHz with 16 GB RAM. In the following three chapters, all benchmarks were tested with all template types and the time step size $\delta \in \{0.1,0.01, 0.001\}$. Timeout (TO) was set to 5 minutes. If a benchmark could be verified as safe, then the corresponding time is entered in the table. If a benchmarks could not be verified as safe, then the entry is (-1).

4.2 Results for HyPro Template Polyhedra

Results. The table with all verification results can be found in the appendix A.1. The Table 4.2 is a summarized version of these results for HTP. Looking at Table 4.2 one can see that the building benchmark could not be verified with the template shapes OI, II and IIG no matter which step size has been chosen, which is expected as adding invariants, guards and bad states should do nothing as the building benchmark does not have invariants or guards; the only bad state constraint it has is already in the OI template shape. Only the with the OCT template shape at least timeouts have been achieved with all step sizes. This is due to the OCT template shape generating 5000 template constraints which increases the solving time of each LP. Similarly, navigation09, platoon30 and platoon42 were all not verifiable with all template shapes and all time steps.
The bouncing ball benchmark could be verified with all shapes with time steps $\delta = 0.01$ and $\delta = 0.001$. One can recognize that the running times do not differ much between each shape with an exception of the OCT shape. This phenomenon can be explained insofar as adding the invariants, guards or bad states does not change the template shape after removing duplicate constraints in this benchmark. This means all intersections could already be efficiently carried out using OI and should not get faster using II, IIG or IIGB. So in theory all running times should be the same for this benchmark. This phenomenon does not occur in general.
With $\delta = 0.01$ and shape OI to IIGB the bouncing ball could not be verified. This is due to the wrapping effect described in Section 2.3: Since adding all the constraints to

Benchmark	**Template Shape**	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$	$\delta = \mathbf{0.001}$
BouncingBall	OI	−1	0.28	1.37
	II	−1	0.28	1.37
	IIG	−1	0.29	1.37
	IIGB	−1	0.33	1.58
	OCT	0.15	0.40	2.68
Building	OI to IIGB	−1	−1	−1
	OCT	−1	TO	TO
navigation03	OI	1.55	11.56	87.78
	II	1.53	11.52	88.79
	IIG	1.54	11.46	107.58
	IIGB	1.53	12.23	107.13
	OCT	11.19	54.68	TO
navigation04	OI	1.77	6.88	76.22
	II	1.78	6.86	75.75
	IIG	2.05	7.28	61.40
	IIGB	1.77	8.08	74.22
	OCT	8.99	52.21	TO
navigation09	OI to IIGB	−1	−1	−1
	OCT	−1	−1	TO
Platoon30	all shapes	−1	−1	−1
Platoon42	all shapes	−1	−1	−1
SpaceRendezvous	OI to IIGB	−1	−1	−1
	OCT	13.24	129.19	TO

Figure 4.2: Running times in seconds for HyPro template polyhedra (HTP). The meaning of the abbreviations of the template shapes can be found in Section 4.1, timeouts are marked with "TO", cases in which safety could not be proven are marked with "-1".

the bouncing ball still results in box shaped constraints, the flowpipe of the bouncing ball has to be overapproximated with these box constraints. The overapproximation error gets amplified with each new segment; in the end the boxes are so big that they intersect the bad states. This happening is depicted in Figure 4.3.
It is apparent that the OCT shape definitely needs a higher running time in the bouncing ball benchmark, up to a factor of two using $\delta = 0.001$. The increase in running time stems from the increased template size, which increased from $|H| = 4$ to $|H| = 8$. Half of the directions are not important for the operations, but still need to be carried along and thus become overhead. This template however affects the precision positively: Even with $\delta = 0.1$ the bouncing ball benchmark could be verified since the octagon template can reduce the impact of the wrapping effect that was apparent with the box constraints of the former shapes. This is also depicted in Figure 4.3.

Both navigation benchmarks behaved very similarly: Naturally, independent of the template shape, the running time increases when the time step size is decreased and

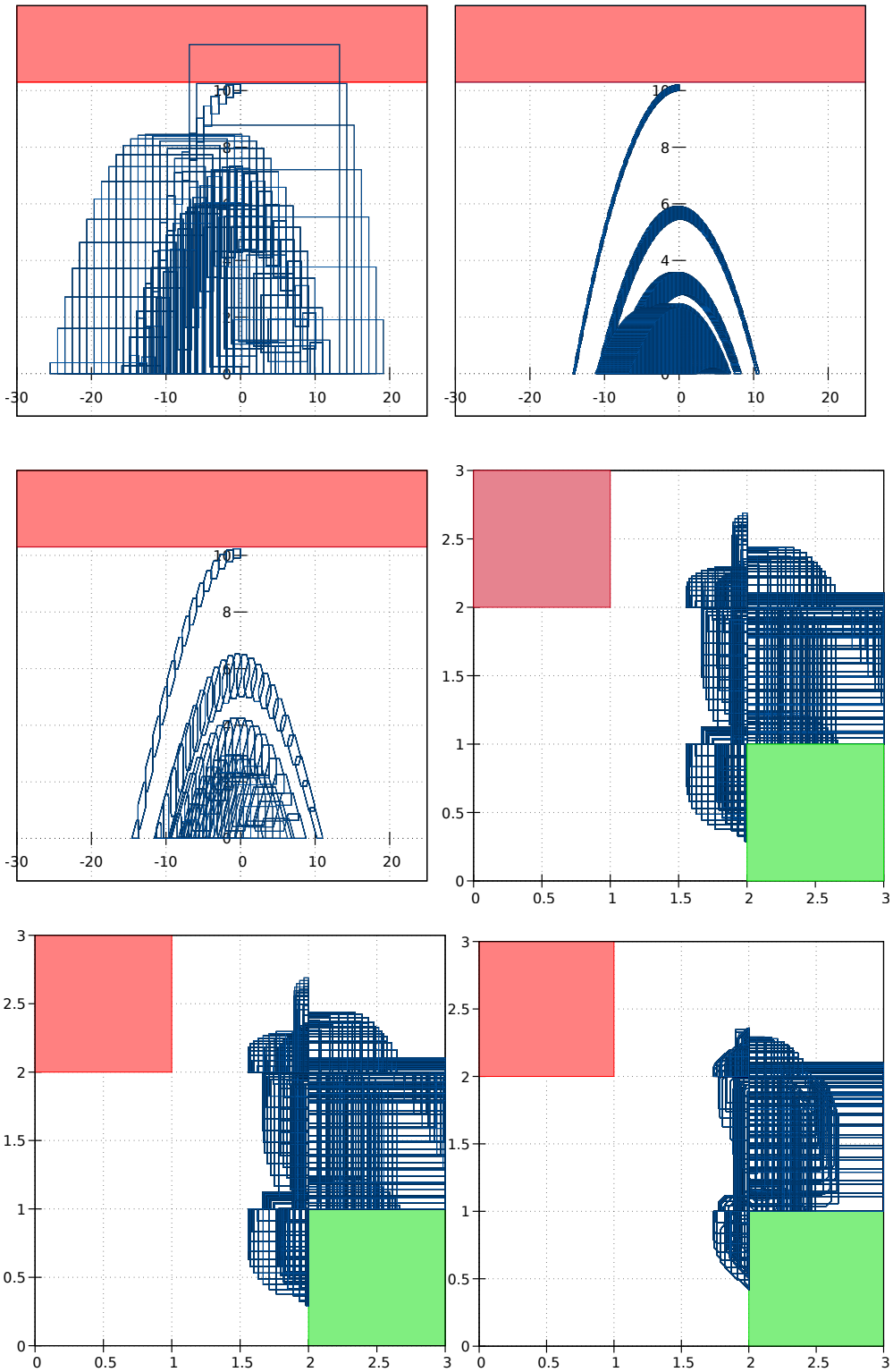


Figure 4.3: Bouncing ball (BB) and navigation03 (nav03) benchmark. **Upper Left:** BB with OI and $\delta = 0.1$ **Upper right:** BB with OI and $\delta = 0.01$ **Middle Left:** BB with OCT and $\delta = 0.1$ **Middle right:** nav03 with OI and $\delta = 0.1$ **Lower left:** nav03 with IIGB and $\delta = 0.1$. **Lower right:** nav03 with OCT and $\delta = 0.1$ Red regions mark bad states, green regions mark good states. The plots for nav03 were made with only 3 jumps instead of 6 jumps.

more segments are computed as a result. Adding already contained constraints like the invariant, guards and bad state constraints did not lead to an increase in precision in navigation03 and navigation04, as effectively no new constraints were added to the template.
Again, the OCT shape needs more running time as the template size is increased. This affects the navigation benchmark insomuch as running time increases, for instance sixfold for the navigation03 benchmark from 13.95 s to 86.78 s for $\delta = 0.1$ and $\delta = 0.01$. At $\delta = 0.01$ one can also see that the high computational time of the affine transformation starts to affect the running time more than the intersections. The running times for II, IIG and IIGB are almost the same with this time step. Interestingly, although the HTP accomplish to verify navigation03 and navigation04 in the most instances, they are not able to verify navigation09 due to its inprecision even with OCT templates. The space rendezvous benchmark on the other hand demonstrates that the increase in precision when using the OCT template is non negligible, as only with the OCT shape this benchmark could be verified. Although it is a benchmark where the addition of invariants, guards and bad state constraints actually enriches the template by new directions, their addition cannot increase the precision enough to verify this benchmark.

Conclusions. The time step size remains the biggest factor for the running times and the precision of the flowpipe computations. The HTP can in theory reduce their running times by adding invariants, guards and bad states. Using the octagon template increases precision in exchange for higher running times, but the gain in precision is not enough to enable the HTP to verify more precision-demanding benchmarks such as platoon42 or building. Nevertheless, in comparison to the H-polytopes, whose running times can be found in Appendix A, the HTP can verify more in half amount of time.

4.3 Results for Taylor Template Polyhedra

At first we will elucidate the effects of the Taylor approximation order needed for the first segment computation and time evolution on the speed and precision of the Taylor approximation-based template polyhedra (TTP). After that the benchmark results are displayed.

Derivative Order. During the Sankaranarayanan reachability analysis, the Taylor approximation needs to be computed to a fixed derivative order DO with the order of the remainder being $DO + 1$. We evaluated the impact of the derivative order DO on the benchmarks bouncing ball, navigation03 and navigation04 with IIGB shaped TTP and $\delta = 0.1$. The results can be seen in Table 4.4.

The first interesting result are the verification results for $DO = 1$: None of the benchmarks could be verified. This is due to the chosen derivative order, which at $m = 1$ only expands the Taylor approximation to the first term and the remainder:

$$H_i x(t + \delta) = H_i x(t) + H_i^{(1)} x(t + \theta)t$$

This equation approximates the trajectory with a straight line with slope $H_i^{(1)} x(t + \theta)$ and intercept $H_i x(t)$ explaining the straightness of each segment, which is depicted in Figure 4.5a for the case of navigation03.
We have seen from Example 3.3.2 that the Lie derivatives of the template rows of the

Benchmark	**DO = 1**	**DO = 2**	**DO = 3**	**DO = 5**
BouncingBall	-1	0.1397s	0.120062s	0.121698s
navigation03	-1	4.78299s	4.94942s	5.46941s
navigation04	-1	1.89089s	6.50195s	4.08465s
Benchmark	**DO = 7**	**DO = 10**	**DO = 20**	**DO = 30**
BouncingBall	0.120132s	0.125093s	0.120814s	0.120895s
navigation03	4.91271s	6.14801s	5.16469s	5.45513s
navigation04	4.21513s	4.80615s	14.5389s	83.0349s

Figure 4.4: Derivative Order Results. All the tests have been conducted with IIGB shape and $\delta = 0.1$.

bouncing ball model vanish for every order greater two. Therefore, any derivative order greater than two does not affect the computation in any way; in the implementation we stop the computation of the Lie derivatives if a Lie derivative of zero is detected. This phenomenon explains the continuously similar running times from $DO = 3$ onwards. In contrast to the bouncing ball the benchmarks navigation03 and navigation04 do not have Lie derivatives that vanish after a certain order. Here navigation04 is exemplary for the general case: The more Lie derivatives need to be computed the larger the running times become. A steady increase in running time can be observed from $DO = 5$ to $DO = 10$. If increased excessively, the computation of the Lie derivatives can take over a main portion of the computation time, here for instance at $DO = 20$ and $DO = 30$, where the running time suddenly increases by a factor of approximately three in comparison from $DO = 10$ to $DO = 20$ and by a factor of 5.7 from $DO = 20$ to $DO = 30$. Incrementing the order does in fact increase precision, but the effect decreases with increasing order. The reason is that every j-th coefficient $\frac{H_i^{(j)}}{j!}\delta^j$ of the Taylor approximation in the time evolution is scaled down by δ^j, and if $\delta < 1$ then $\delta^j \to 0$ when $j \to DO$ for all $j \leq DO$. This decrease in extra precision can be seen in Figure 4.5.

The running time results for navigation03 do not increase in the same way the running time results for navigation04 did. Navigation03 is a corner case insofar as its fixed point revolves around the destination location where no flow is defined. In a location where no flow influences the dynamics of the state sets, the Lie derivative is always zero, and thus we can stop the computation of the Lie derivatives early, which expresses itself in running times continuously below seven seconds even at $DO = 20$. This is not the case with navigation04; it switches back and forth between two grid cells where flow is defined if enough time has passed.

In summary increasing the derivative order increases precision, but less with each order. At the same time it increases running times only slightly for orders below ten. The computation of a Lie derivatives of order DO can be stopped when a Lie derivative of order $j < DO$ is zero, which additionally saves time. In our following experiments we have chosen a derivative order of $DO = 6$ according to the results of these experiments.

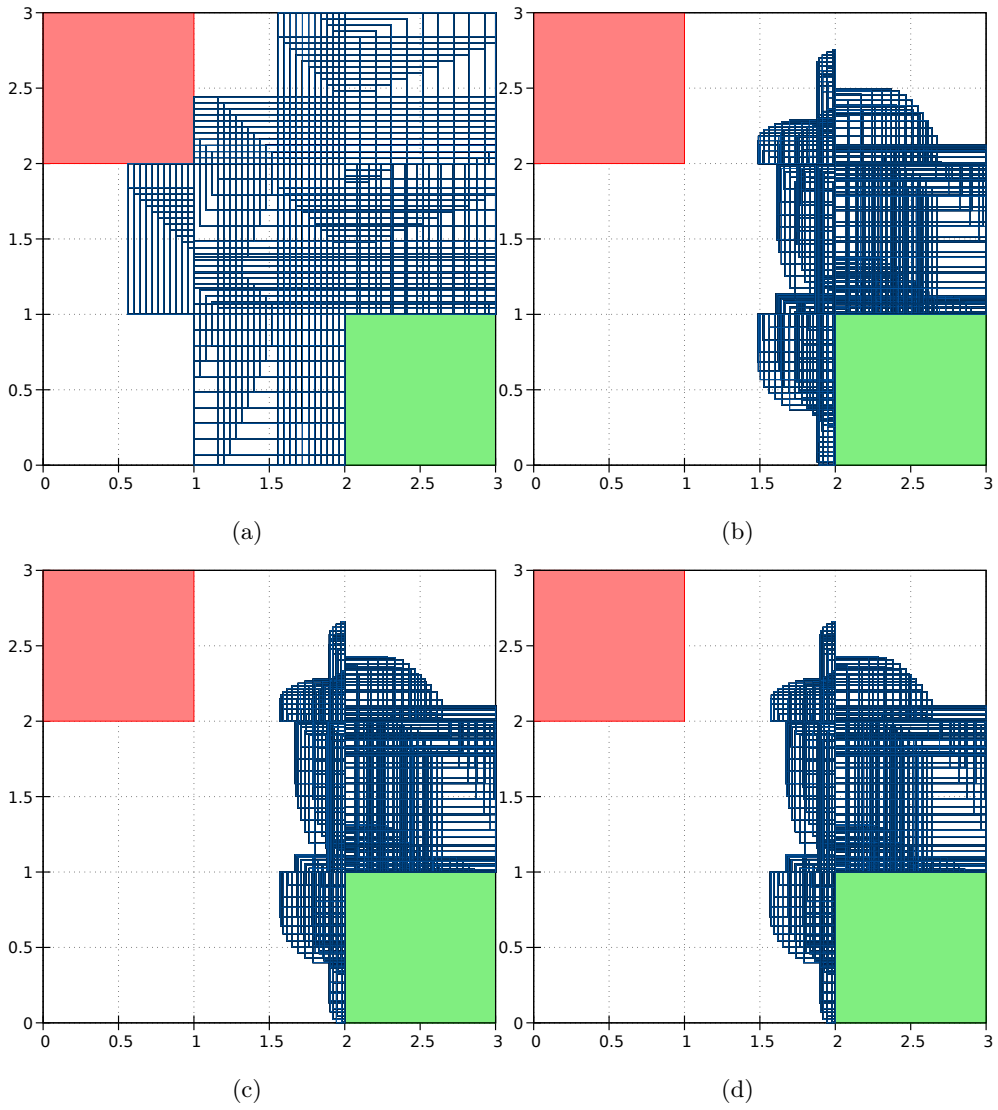


Figure 4.5: Derivative Order plots. All pictures are made with navigation03 with IIGB, $\delta = 0.1$ and 3 instead of 6 jumps. **(a)** $DO = 1$ **(b)** $DO = 3$ **(c)** $DO = 6$ **(d)** $DO = 10$

Benchmark	Template Shape	$\delta = 0.1$	$\delta = 0.01$	$\delta = 0.001$
BouncingBall	OI	−1	0.29s	1.49s
	II	−1	0.29s	1.49s
	IIG	−1	0.28s	1.74s
	IIGB	−1	0.28s	1.47s
	OCT	0.12s	0.56s	3.45s
Building	OI to IIGB	−1	−1	−1
	OCT	TO	TO	TO
navigation03	OI	5.53s	34.30s	TO
	II	5.95s	34.29s	TO
	IIG	5.86s	41.11s	TO
	IIGB	5.79s	34.65s	TO
	OCT	73.95s	TO	TO
navigation04	OI	4.50s	28.10s	237.55 s
	II	3.78s	23.06s	240.37 s
	IIG	3.77s	27.89s	238.40 s
	IIGB	3.74s	23.13s	237.50 s
	OCT	44.46s	TO	TO
navigation09	OI to IIGB	−1	−1	−1
	OCT	−1	−1	TO
Platoon30	OI too IGB	−1	−1	−1
	OCT	−1	−1	TO
Platoon42	OI to IIGB	−1	−1	−1
	OCT	−1	−1	TO
SpaceRendezvous	OI to IIGB	−1	−1	−1
	OCT	−1	TO	TO

Figure 4.6: Benchmark results for Taylor approximation-based template polyhedra (TTP). All tests were conducted with Taylor approximation order $DO = 6$. The full table is visible in appendix A.

Benchmark Results. All tests were conducted with a Taylor approximation order of $DO = 6$ and location invariant strengthening to counter unboundedness of the invariants. Overall the TTP benchmark results in Table 4.6 look similar to the HTP results. Just as the HTP, neither building, navigation09, platoon30 nor platoon42 could be verified, but in contrast to the former, more timeouts were reached. This time space rendezvous could not be verified with any shape and any time step size. The OCT shape lead to timeouts for instance in navigation09, platoon30, platoon42 and space rendezvous with $\delta = 0.001$. There the reason behind the timeouts is the increased precision and increased running times finer time steps bring with them. In the case of the building benchmark with the OCT template shape, the same reason as for the HTP, namely the strong increase in constraints and the consequently high running times for each LP call is the reason why timeouts are observed.
For $\delta = 0.1$ the bouncing ball benchmark could not be verified with the OI, II, IIG and IIGB shape, but with the OCT shape since the octagon template increases precision.

The running times for the bouncing ball benchmark with HTP with the template shapes OI, II, IIG and IIGB using the finest time step size $\delta = 0.001$ are between 7% faster (with IIGB) to 27% slower (with OI) than the TTP using the same shape with an average of being 9.5% slower, but since the differences in running time are within an interval of 0.4 seconds, these comparisons could be influenced by external disturbances. The same holds true for $\delta = 0.01$.
Similar to the HTP, navigation03 and navigation04 could be verified with $\delta = 0.1$ and $\delta = 0.01$ with any shape except in the case of the OCT template at $\delta = 0.01$. For navigation03 and $\delta = 0.1$ the TTP turn out to be on average about three times (276%) slower than the HTP using the same shape. For the same benchmark and $\delta = 0.01$ they are on average slower by a factor of two (209%). This difference in speed is also noticeable in the navigation04 benchmark: For $\delta = 0.1$ the TTP are on average 115% slower than the HTP using any shape except OCT, for $\delta = 0.01$ 235% slower and for $\delta = 0.001$ 234% slower. This increase in running time stems from the fact that more segments are computed in Sankaranarayanan method, for instance in navigation03 with IIGB and $\delta = 0.1$ the Sankaranarayanan method generates 1891277 segments, while the HTP generate only 592183 segments with the same parameters, which are only 31% of the first amount of segments. Many of the segments additionally generated are part of flowpipes leading to fixed points, which come from transitions that were additionally enabled through overapproximation.
Another cause for the higher running times is that in the first segment computation the roots of the univariate polynomial $p_i(t)$ need to be computed. There exists a variety of sophisticated root finding or root approximation algorithms, whose inner workings are beyond the scope of this thesis. During implementation, it became apparent that the underlying root finding algorithm is costly as it takes about 20% of the running time. Therefore it is sensible to find a procedure that can avoid the root computation without abandoning the soundness of the approach. An idea of a possible approach is sketched in Chapter 5.
The resulting plots are similar to the plots of the HTP, which can be seen in Figure 4.7, but visibly contain multiple flowpipes for instance in Figure 4.7d. In the same picture one can see that the flowpipe in the middle square is more precise than in Figure 4.7c. On the other hand there is an extra flowpipe in Figure 4.7b left of the green region, which is not existent in Figure 4.7a.

Conclusions. The TTP combine multiple different techniques into one, which one can tweak at several parameters to obtain the best possible result. Next to the choice of the template, which is just as crucial for the TTP as it is for the HTP, the derivative order plays an important role in the precision. In benchmarks where the Lie derivative does not vanish at the j-th order with $j < DO$, it can increase precision at a low running time addition. Albeit being slower than the HTP, with the usage of location invariant strengthening both implementations compare similar in precision. The TTP surpass the H-polytopes in running time and positive verifications.

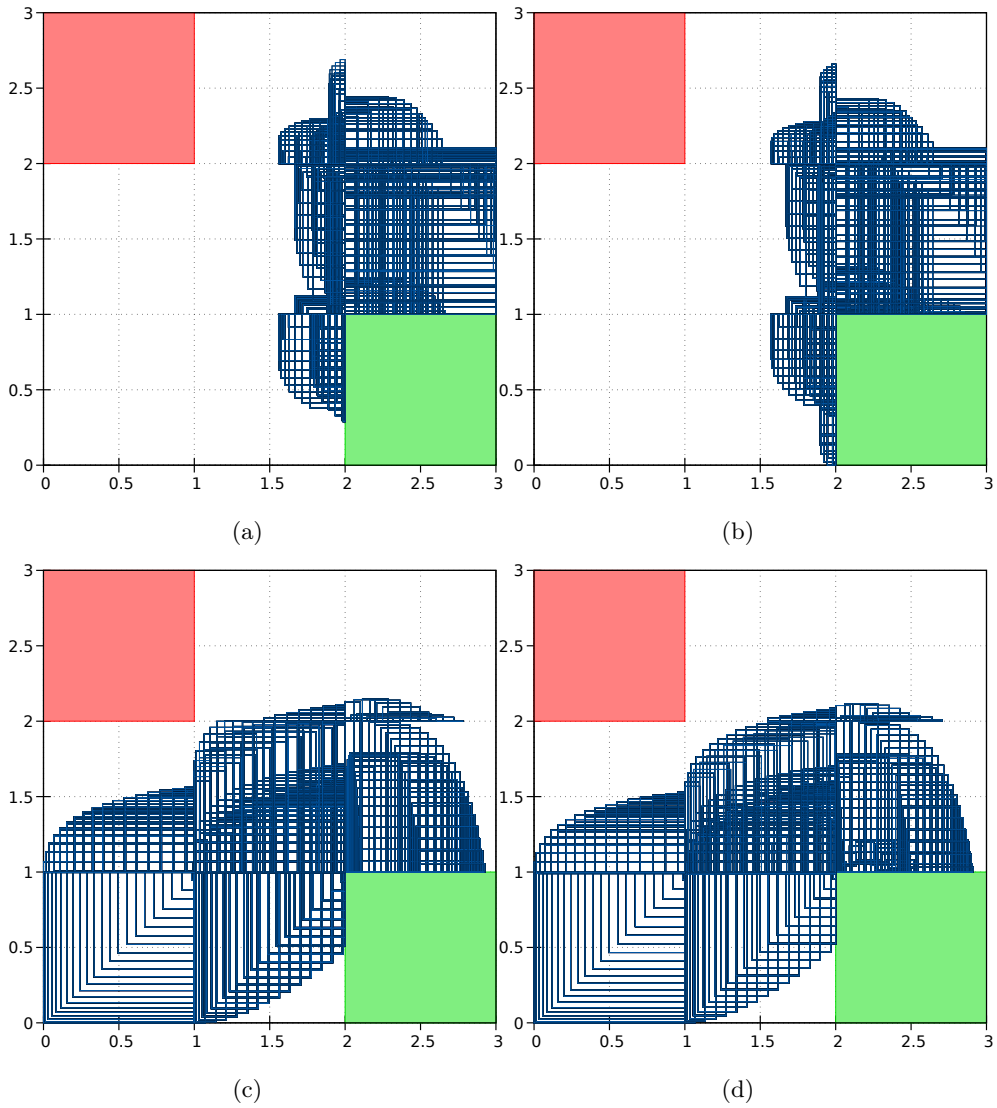


Figure 4.7: navigation03 (Nav03) and navigation04 (Nav04). All pictures done with IIGB shape, $\delta = 0.1$ and three jumps. **(a)** Nav03 HTP. **(b)** Nav03 TTP. **(c)** Nav04 HTP. **(d)** Nav04 TTP.

4.4 Results for Support Functions with HTP

Benchmark	Template Shape	$\delta = 0.1$	$\delta = 0.01$	$\delta = 0.001$
BouncingBall	OI	0.12s	0.18s	0.99s
	II	0.12s	0.18s	0.94s
	IIG	0.12s	0.21s	1.07s
	IIGB	0.12s	0.20s	1.08s
	OCT	0.12s	0.18s	0.82s
Building	OI	−1	−1	2.55s
	II	−1	−1	2.26s
	IIG	−1	−1	2.35s
	IIGB	−1	−1	2.07s
	OCT	−1	−1	2.05s
navigation03	OI	39.36	TO	TO
	II	36.29s	TO	TO
	IIG	44.59s	TO	TO
	IIGB	36.29s	TO	TO
	OCT	36.27s	TO	TO
navigation04	all shapes	−1	−1	TO
navigation09	all shapes	−1	−1	−1
Platoon30	OI	4.60s	6.43s	22.62
	II	4.60s	6.23s	21.97s
	IIG	4.61s	6.24s	18.16s
	IIGB	5.35s	6.84s	19.18s
	OCT	5.17s	6.22s	21.89s
Platoon42	OI	5.50s	6.33s	20.22
	II	4.59s	6.22s	18.40s
	IIG	4.60s	6.23s	18.15s
	IIGB	5.30s	7.33s	20.11s
	OCT	4.58s	6.22s	18.13s
SpaceRendezvous	OI	0.88s	13.98s	TO
	II	0.90s	11.06s	TO
	IIG	0.85s	12.86s	TO
	IIGB	0.85s	12.29s	TO
	OCT	0.84s	12.76s	TO

Figure 4.8: Benchmark Results for Support Functions with HTP.

Results. The results for support functions in conjunction with HTP as underlying state set representation can be seen in Table 4.8. Most apparent are the successful verification results for bouncing ball, building (only with $\delta = 0.001$), platoon30, platoon42 and space rendezvous (with $\delta = 0.1$ and $\delta = 0.01$). Just as apparent are the results for navigation03, navigation04 and navigation09: While navigation04 and navigation09 could not be verified with any shape and any time step, navigation03 could be proven as safe with $\delta = 0.1$ and reached timeout for $\delta = 0.01$ and $\delta = 0.001$. Overall the

results are similarly structured as the results from support functions with H-polytopes as underlying representations visible in Appendix A, which was to be expected since template polyhedra behave like H-polyhedra with regards to maximization as both solve LPs for this operation, but template polyhedra can avoid LP solver calls if the maximization direction is a template direction.
The bouncing ball benchmark achieved similar running times with any shape within one time step size and stands on the same level as the running times of the support functions with H-polytopes. The small decrease in running time with the OCT template could stem from the increased size of the template, which in turn increases the chance that a maximization direction is equal to a template row leading to more cases where the efficient maximization operation of the template polyhedra is activated. The running times of the bouncing ball benchmark with any shape and $\delta = 0.001$ using HTP alone are between 28% (with IIGB) and 46% (with OI) and on average 40% (without OCT) slower than support functions with the template polyhedra and even 326% slower using OCT template. This phenomenon can be traced back to several optimizations made for the support functions, one being for instance that the underlying representation is converted into a box if the template constraints are found out to resemble a box shape anyway, for whom computing the support into any direction and not only the template directions can be done efficiently.
As already mentioned the building benchmark could be solved with $\delta = 0.001$. An insight gained from this result is that the precision of the support functions with template polyhedra is enough to solve precision-demanding benchmarks such as the building benchmark. With this in mind, it seems plausible that platoon42, its more difficult version platoon30 and space rendezvous (only with $\delta = 0.1$ and $\delta = 0.01$) were also verified. During computation, the support functions are not limited in precision by the wrapping effect as the approximating halfspace normals are dynamically transformed into different directions by the support function operation rules, in contrast to the template polyhedra alone, whose halfspace directions are fixed.
The support function needs to make multiple time-consuming traversals to compute set emptiness as discussed in Section 2.4. Since during each intersection with the invariants, guards and bad states the emptiness of the result must be computed, support functions have high running times for benchmarks where intersections and emptiness need to be computed often, here for instance navigation03, navigation04 and navigation09. This explains the running times for $\delta = 0.1$ and the timeouts for the finer time steps. The plots can be seen in Figure 4.9. The overapproximation error on these plots is large and is increased with every jump. Although the reason for this remains to be investigated, the intersection with the guards and the following reset could play a role.

Conclusions. Support functions with template polyhedra can verify more benchmarks than HTP and TTP. Even precision-demanding benchmarks can be verified with them. As the support functions are not as efficient with intersections and emptiness as the template polyhedra, the benchmarks that are verified by the support functions tend to branch less and therefore need to calculate less segments for whom intersection and emptiness need to be computed. The template polyhedra in return thrive at benchmarks with high amounts of intersections.

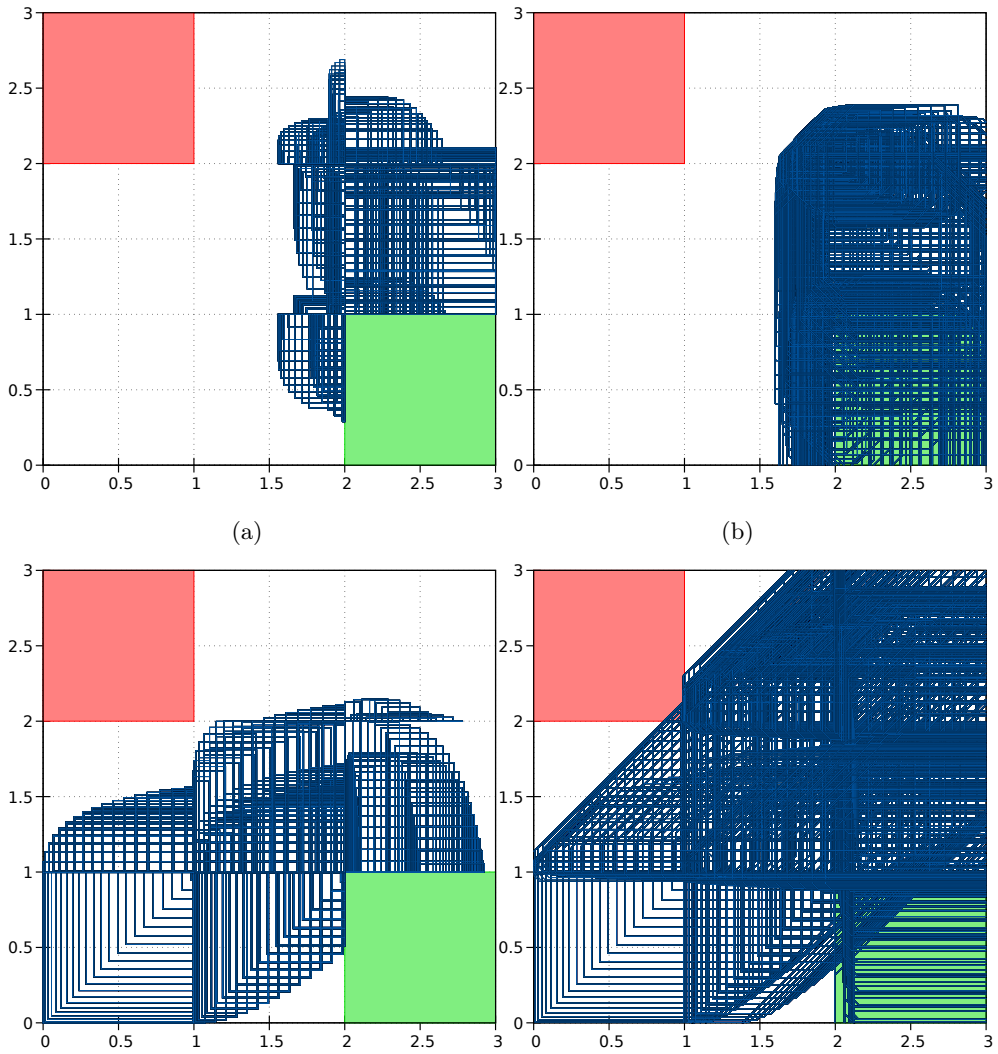


Figure 4.9: Support function benchmark results. All plots done with $\delta = 0.1$, IIGB shape and 3 instead of 6 jumps. **(a)** nav03 with HTP **(b)** nav03 support functions with HTP **(c)** nav04 with HTP **(d)** nav04 support functions with HTP

4.5 Further Optimization Results

In our previous tests we tested the HTP and TTP on various benchmarks and tabularized their running times. To further increase the efficiency of the template polyhedra, the fixed point detection tests (short FIX) described at the end of Section 3.4 were also tested in conjunction with LIS. At last, the effects of the numerical corrections and the cycle detection (both abbreviated with NC) from Section 3.4 in LIS are checked. The benchmarks used in this test series are the navigation instances one to eight, where every instance has a time bound of three and a maximum number of jumps of six. The navigation benchmarks were chosen for their high amount of fixed points. This time six different combinations are tested: Each of the two template shapes IIGB and OCT are tested standalone, with the fixed point detection in addition and lastly with a combination of the fixed point detection and the numerical corrections. Each of these strategies are tested with $\delta = 0.1$ and $\delta = 0.01$.

Results. Since the table of benchmark results has too many entries, we decided to display it as Appendix A. The fixed point detection procedure reduces the running time of every benchmark without the fixed point detection independent of the template shape by a significant amount. In some instances, for example for navigation02 with OCT shape and $\delta = 0.01$, it reduced the running time so much that the test was not interrupted and terminated by the timeout. This large decrease in running time stems from the fact that the navigation benchmarks all have fixed points that are reached within the time bound. These fixed points are detected and the flowpipe computation can be avoided. The plots consequently show less segments, which can be seen in Figure 4.10. As fixed point detection does not increase precision, only benchmarks that could be verified without fixed point detection can be verified with fixed point detection.
The effect of the numerical corrections and the cycle detection depend on the benchmarks and the template shapes. In navigation01 to navigation04 using IIGB, fixed point detection with numerical corrections and cycle detection reduced running times by an average of 8% using $\delta = 0.1$ and 16% using $\delta = 0.01$ in comparison to IIGB with only fixed point detection. In navigation07 with IIGB it even increased running times slightly. With the OCT shapes we can see a similar effect: The running times using OCT+FIX+NC are decreased with either time step in the first four navigation instances but are increased in the latter three instances apart from navigation07 with $\delta = 0.1$ where it decreased.
The cycle detection and the numerical corrections shrink the number of iterations needed until a fixed point is found, which explains the running time reduction in the first four benchmarks. The numerical corrections however cannot avoid all cases where solving the LP for the next strengthened invariant returns infeasibility. The cause of this phenomenon and the running time increase in the latter benchmarks has not been found and remains to be investigated. The effects on precision from the usage of the fixed point detection and the numerical corrections are depicted in Figure 4.10.

Conclusions. While the fixed point detection greatly reduces running time in benchmarks with fixed points by avoiding subsets of already computed flowpipes, the cycle detection and the numerical corrections can influence the running times positively in smaller benchmarks.

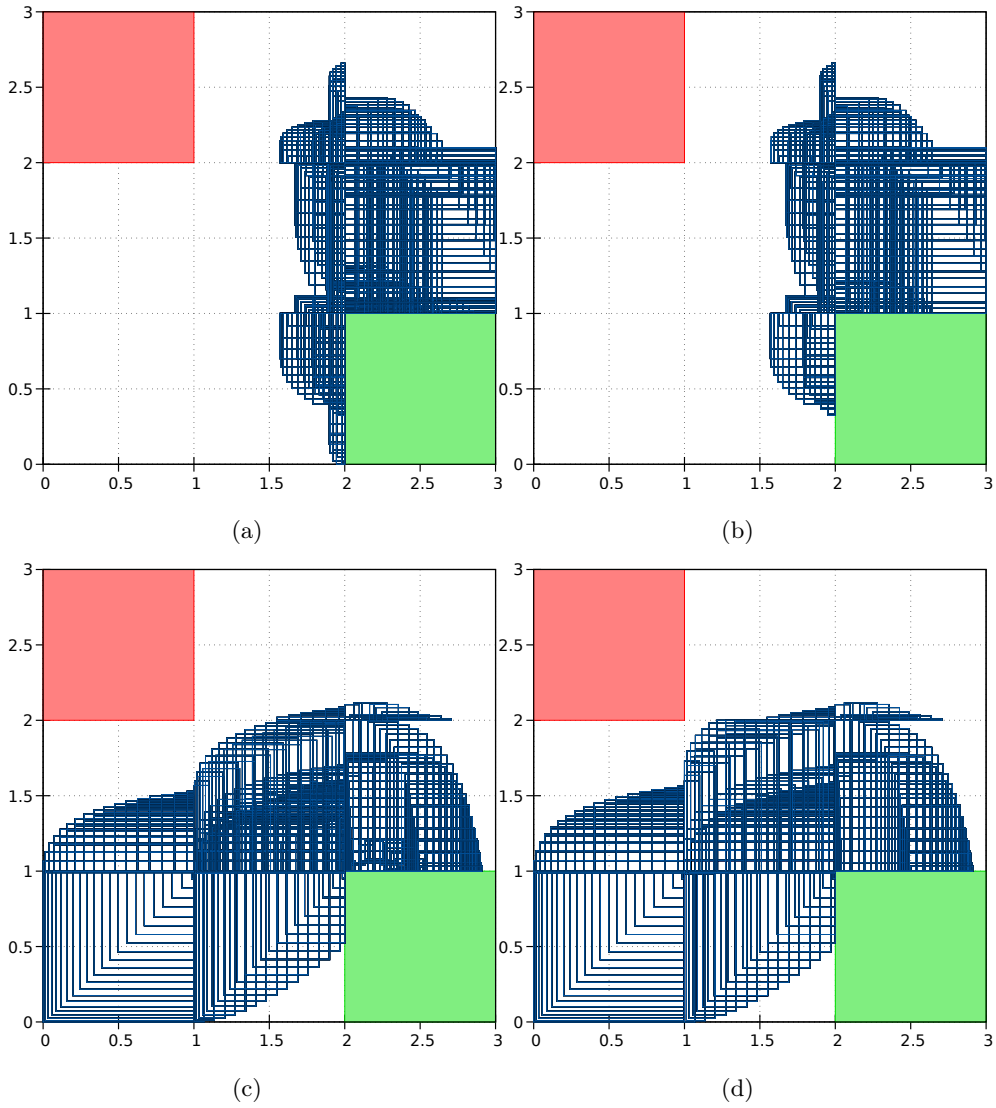


Figure 4.10: Navigation03 and navigation04. All pictures made with $DO = 6$, $\delta = 0.1$ and three jumps. **(a)** nav03 TTP **(b)** nav03 TTP with fixed point detection and numerical corrections **(c)** nav04 TTP **(d)** nav04 TTP with fixed point detection and numerical corrections. Notice that the counterparts with fixed point detection are cleaned from inner flowpipes.

Chapter 5

Conclusion

Summary. Over the course of this thesis we gave an overview on formal verification of hybrid systems via flowpipe construction and took a deeper look at two state set representations: Template polyhedra and support functions. With respect to their usage in the reachability analysis of affine hybrid automata, we studied their properties with special focus on the efficient operations of both representations. For template polyhedra two approaches regarding efficient flowpipe-construction-based reachability analysis were presented and implemented: A version following the general flowpipe construction through the provided interface from HyPro and a Taylor approximation-based method proposed by Sankaranarayanan et al. that calculates the first segment and the time evolution through overapproximating the Taylor approximation of the trajectory of each template row. Both approaches were evaluated on several benchmarks, some of which require frequent intersection operations with differences in running times; the HyPro implementation of the template polyhedra was consistently faster and could only be matched by the Taylor approximation-based template polyhedra when fixed point detection, cycle detection and numerical corrections were added.
The content of the templates strongly effects the running times and precision of the template polyhedra during reachability analysis. Different template shapes, for example only the initial constraints, all constraints that are known within all locations or octagonal constraints also yielded different results: In general, having all constraints, namely the invariants, guards of outgoing transitions and possible bad state constraints in the template led to faster verification results whereas the octagonal templates increase precision. Similar tests conducted on support functions in combination with the template polyhedra suggest that template polyhedra and support functions complement each other with respect to the efficiency of certain operations, i.e. intersection since the benchmarks that were solved by the support functions with great precision in a short amount of time are those that the template polyhedra could not verify amongst others.
In addition to these benchmark tests, we experimented with several optimizations, the biggest one of which is the location invariant strengthening. Its ability to tighten the invariant bounds with respect to the initial set and the given invariants under consideration of the local flow renders it especially useful in the Taylor approximation-based method, where it increases the precision and speed of the computation as a consequence. With this technique we developed a method to use the Taylor approximation-based approach on unbounded invariants and refined it with cycle detection to identify cycles

within the policy iteration and numerical corrections to avoid inconsistency within the location invariant strengthening. In addition to location invariant strengthening, fixed point detection has proven itself useful in reducing running times by avoiding the recomputation of flowpipes that are a subset of already computed flowpipes. The fixed point detection can be efficiently performed with template polyhedra, due to its efficient subset operation. All these techniques combined result in an efficient state set representation that has a lot of potential for further research.

Future Work. The future work on template polyhedra includes the following topics:

Bounding Heuristics. One topic of further research includes location invariant strengthening with unbounded invariants as in general this technique works purely on bounded invariants. Although our method to artificially bound the feasible invariant region with unattainable high numbers works, the current estimation of these bounds is based on empiric experiences with the respective model. One possibility in solving this could be to simulate at least one state beforehand and derive the smallest unreachable bounds from the simulation, but an efficient procedure to automate the choice of these bounds needs to be researched. An additional disadvantage of our current method is its incompatibleness with unbounded time verification as in this scenario the valuations of the variables are not bounded by the time bound.

Root Computation Optimization. As mentioned in Section 3.3 the computation of the roots of the polynomial $p_i(t)$ that overapproximates the trajectory in the Sankaranarayanan method is costly and should therefore be avoided if possible. Since $p_i(t)$ is an univariate polynomial of degree DO of whom we seek the roots between the interval $[0,\delta]$ an interval-based method like interval constraint propagation could be used to determine whether a root lies in $[0,\delta]$ and within what boundaries it lies [Dav87]. Whether this approach is faster than the analytical root computation remains to be researched.

Template Refinement. Most crucial is the choice of the template when working with template polyhedra since this decision affects speed and precision of the template polyhedra greatly. More experiments with different template shapes and their combination need to be conducted. Sankaranarayanan et al. for instance suggested the inclusion of the Lie derivatives of every row or the eigenvalues of the template [SDI08b]. Another idea deals with dynamic modification of the template during the flowpipe computation: In a counterexample-guided refinement approach spurious counterexamples could be used to modify the template such that the counterexample cannot be reached anymore as done in [BFGH17].

Duality. Our experiments show that template polyhedra and support functions supplement each other. While template polyhedra are particularly beneficial for the running times of benchmarks with many intersections, support functions in contrast are more successful in all other benchmarks. A state set representation that can exploit the properties of both depending on the currently requested operation needed could increase efficiency even more. This idea has been fleshed out in [FLGD$^+$11].

Regular polytopes. One idea that came to mind during experiments with the octagonal template was the idea of using a different representation as the underlying representation of the support functions; a representation that can compute the support quickly into any direction. This idea led to *regular polytopes*, which are a polytopes of dimension d whose facets are regular in $d-1$. This definition recursively defines regularity until $d = 2$, where a polygon (a polytope in dimension two) is regular if all its interior angles and edge lengths are the same. In a regular polygon every maximization direction is guaranteed to lie in one of the cones that are defined by two neighbouring facet normals. These two neighbouring facet normals and their respective halfspaces are enough to uniquely determine their intersection point or rather the point which gives the optimal solution to the underlying maximization problem in the maximization direction. This means that if one can find out in which cone the maximization direction lies in, then the optimal point can be inferred by solving a linear program over only two constraints. We claim that the question remaining, namely how to calculate in which cone the maximization direction l lies in, can be solved by finding the two facet normals n, n' that are the "closest" to the maximization direction: By computing the normalized scalar products $(\frac{l}{||l||})^T \frac{n_i}{||n_i||}$ for all normals n_i in the set of all facet normals and then selecting the two normals n,n' where the scalarproduct is maximized, the two halfspaces determining the optimal point for l can be found. Our claim went even further: We conjectured that in any dimension d, if we have a regular polytope $P = \bigcap_{i=0}^{m-1} n_i x \leq c_i$ in $\mathbb{R}^d$ with normalized normals n_i from $N = \{n_0,...,n_m\}$ and a normalized direction l, then the d normals with the highest value of $l^T n$ make up the cone where l lies in. Thus, only the d halfspaces defined by these normals are needed for finding the optimal value. A formal proof of this claim is still required and subject of ongoing research.
If used in conjunction with the support functions, the regular polytopes could store their extreme points once computed. This enables a fast lookup procedure where only m dot products have to be computed for maximizing into a given direction. Additionally, since as an underlying representation the actual regular polytope would not be modified in any way, regularity is preserved for each flowpipe computation once established. If used alone, we predict that reestablishing regularity for each segment will be a computational burden that may be solved using template polyhedral techniques such as the overapproximation that have been studied in this thesis.

Bibliography

[Ábr16] E. Ábrahám. Modelling and analysis of hybrid systems, 2016. Lecture Notes RWTH Aachen.

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

[AD94] Rajeev Alur and David L Dill. A theory of timed automata. *Theoretical computer science*, 126(2):183–235, 1994. Elsevier.

[Ber11] Dimitri P Bertsekas. Dynamic programming and optimal control 3rd edition, volume ii. *Belmont, MA: Athena Scientific*, 2011.

[BFGH17] Sergiy Bogomolov, Goran Frehse, Mirco Giacobbe, and Thomas A Henzinger. Counterexample-guided refinement of template polyhedra. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 589–606. Springer, 2017.

[Bla99] Franco Blanchini. Set invariance in control. *Automatica*, 35(11):1747–1767, 1999. Elsevier.

[Bra05] Michael S Branicky. Introduction to hybrid systems. In *Handbook of networked and embedded control systems*, pages 91–116. Springer, 2005.

[BV04] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[CKJ+15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT : an Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Theory and Applications of Satisfiability Testing : SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 360–368. International Conference on Theory and Applications of Satisfiability Testing, Austin, Tex. (USA), 24 Sep 2015 - 27 Sep 2015, Springer International Publishing, Sep 2015.

[CM17] Nicole Chan and Sayan Mitra. Verifying safety of an autonomous spacecraft rendezvous mission. *arXiv preprint arXiv:1703.06930*, 2017.

[CS11] Michael A Colón and Sriram Sankaranarayanan. Generalizing the template polyhedral domain. In *European Symposium on Programming*, pages 176–195. Springer, 2011.

[CSM+15] Xin Chen, Stefan Schupp, Ibtissem Ben Makhlouf, Erika Ábrahám, Goran Frehse, and Stefan Kowalewski. A benchmark suite for hybrid systems reachability analysis. In *NASA Formal Methods*, pages 408–414, Cham, 2015. Springer.

[CVD02] Younes Chahlaoui and Paul Van Dooren. A collection of benchmark examples for model reduction of linear time invariant dynamical systems. 2002. Niconet.

[Dav87] Ernest Davis. Constraint propagation with interval labels. *Artificial intelligence*, 32(3):281–331, 1987. Elsevier.

[DG11] Thao Dang and Thomas Martin Gawlitza. Template-based unbounded time verification of affine hybrid automata. In *Asian Symposium on Programming Languages and Systems*, pages 34–49. Springer, 2011.

[DH12] Dirk Den Hertog. *Interior point approach to linear, quadratic and convex programming: algorithms and complexity*, volume 277. Springer Science & Business Media, 2012.

[F+04] Komei Fukuda et al. Frequently asked questions in polyhedral computation. *ETH, Zurich, Switzerland*, 2004. https://www.cs.mcgill.ca/ fukuda/soft/polyfaq/polyfaq.html.

[FI04] Ansgar Fehnker and Franjo Ivančić. Benchmarks for hybrid systems verification. In *Hybrid Systems: Computation and Control*, pages 326–341. Springer, 2004.

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.

[For84] Otto Forster. Analysis 2 Differentialrechnung im Rn. *Gewöhnliche Differentialgleichungen, Vieweg Braunschweig*, 1984.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *European symposium on programming*, pages 237–252. Springer, 2007.

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *International Workshop on Hybrid Systems: Computation and Control*, pages 291–305. Springer, 2005.

[GSBS19] Jessica Gronski, Mohamed-Amin Ben Sassi, Stephen Becker, and Sriram Sankaranarayanan. Template polyhedra and bilinear optimization. *Formal Methods in System Design*, 54(1):27–63, 2019. Springer.

[HH94] Thomas A Henzinger and Pei-Hsin Ho. Hytech: The cornell hybrid technology tool. In *International Hybrid Systems Workshop*, pages 265–293. Springer, 1994.

[HKPV98] Thomas A Henzinger, Peter W Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of computer and system sciences*, 57(1):94–124, 1998. Elsevier.

[LG09] Colas Le Guernic. *Reachability analysis of hybrid systems with linear continuous dynamics*. PhD thesis, 2009.

[LGG09] Colas Le Guernic and Antoine Girard. Reachability analysis of hybrid systems using support functions. In *Computer Aided Verification*, pages 540–554. Springer, 2009.

[Mak00] Andrew Makhorin. The gnu linear programming kit (glpk), 2000.

[Min01] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Symposium on Program as Data Objects*, pages 155–172. Springer, 2001.

[MMH+11] Ibtissem Ben Makhlouf, Jan P Maschuw, Paul Hänsch, Hilal Diab, Stefan Kowalewski, and Dirk Abel. Safety verification of a cooperative vehicle platoon with uncertain inputs using zonotopes. *IFAC Proceedings Volumes*, 44(1):9769 – 9774, 2011. 18th IFAC World Congress, Elsevier.

[PS98] Christos H Papadimitriou and Kenneth Steiglitz. *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1998.

[SÁ18] Stefan Schupp and Erika Ábrahám. The HyDRA Tool : A Playground for the Development of Hybrid Systems Reachability Analysis Methods. In *Proceedings of the PhD Symposium at iFM18 on Formal Methods: Algorithms, Tools and Applications (PhD-iFM18)*, volume 483 of *Research report*. Oslo University, Sep 2018.

[SÁBMK17] 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 : 9th international symposium*, volume 10227 of *Lecture Notes in Computer Science*, pages 288–294. NASA Formal Methods (NFM) Symposium, Moffett Field, CA (USA), 16 May 2017 - 18 May 2017, Springer, May 2017.

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

[SDI08a] Sriram Sankaranarayanan, Thao Dang, and Franjo Ivančić. A policy iteration technique for time elapse over template polyhedra. In *International Workshop on Hybrid Systems: Computation and Control*, pages 654–657. Springer, 2008.

[SDI08b] Sriram Sankaranarayanan, Thao Dang, and Franjo Ivančić. Symbolic model checking of hybrid systems using template polyhedra. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 188–202. Springer, 2008.

[TNJ16] Hoang-Dung Tran, Luan Viet Nguyen, and Taylor T Johnson. Large-scale linear systems from order-reduction (benchmark proposal). In *3rd Applied Verification for Continuous and Hybrid Systems Workshop (ARCH), Vienna, Austria*, 2016.

[War13] Frank W Warner. *Foundations of differentiable manifolds and Lie groups*, volume 94. Springer Science & Business Media, 2013.

[Zie12] Günter M Ziegler. *Lectures on polytopes*, volume 152. Springer Science & Business Media, 2012.

Appendix A

Benchmark results

HyPro H-Polytope Benchmark Results:

Benchmark	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$	$\delta = \mathbf{0.001}$
BouncingBall	0.15s	0.47s	3.78s
Building	TO	TO	TO
Navigation03	TO	TO	TO
Navigation04	TO	TO	TO
Navigation09	TO	TO	TO
Platoon30	TO	TO	TO
Platoon42	TO	TO	TO
SpaceRendezvous	TO	TO	TO

HyPro Support Functions with H-Polytopes Benchmark Results:

Benchmark	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$	$\delta = \mathbf{0.001}$
BouncingBall	0.12s	0.22s	1.06s
Building	-1	-1	2.69s
Navigation03	32.80s	295.41 s	TO
Navigation04	-1	-1	TO
Navigation09	-1	-1	TO
Platoon42	5.75s	7.66s	21.61s
Platoon30	5.70s	6.34s	18.30s
SpaceRendezvous	1.12s	20.00s	TO

HyPro Template Polyhedra Benchmark Results:

Benchmark	**Template Shape**	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$	$\delta = \mathbf{0.001}$
BouncingBall	OI	−1	0.28s	1.37s
	II	−1	0.28s	1.37s
	IIG	−1	0.29s	1.37s
	IIGB	−1	0.33s	1.58s
	OCT	0.15s	0.40s	2.68s
Building	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	TO	TO
navigation03	OI	1.55s	11.56s	87.78s
	II	1.53s	11.52s	88.79s
	IIG	1.54s	11.46s	107.58 s
	IIGB	1.53s	12.23s	107.13 s
	OCT	11.19s	54.68s	TO
navigation04	OI	1.77s	6.88s	76.22s
	II	1.78s	6.86s	75.75s
	IIG	2.05s	7.28s	61.40s
	IIGB	1.77s	8.08s	74.22s
	OCT	8.99s	52.21s	TO
navigation09	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	−1	TO
Platoon30	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	−1	−1
Platoon42	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	−1	−1
SpaceRendezvous	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	13.24s	129.19 s	TO

Figure A.1: Benchmark results for HyPro template polyhedra. The meaning of the abbreviations of the template shapes can be found in section 4.1.

Sankaranarayanan Template Polyhedra Benchmark Results:

Benchmark	**Template Shape**	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$	$\delta = \mathbf{0.001}$
BouncingBall	OI	−1	0.29s	1.49s
	II	−1	0.29s	1.49s
	IIG	−1	0.28s	1.74s
	IIGB	−1	0.28s	1.47s
	OCT	0.12s	0.56s	3.45s
Building	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	TO	TO	TO
navigation03	OI	5.53s	34.30s	TO
	II	5.95s	34.29s	TO
	IIG	5.86s	41.11s	TO
	IIGB	5.79s	34.65s	TO
	OCT	73.95s	TO	TO
navigation04	OI	4.50s	28.10s	237.55 s
	II	3.78s	23.06s	240.37 s
	IIG	3.77s	27.89s	238.40 s
	IIGB	3.74s	23.13s	237.50 s
	OCT	44.46s	TO	TO
navigation09	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	−1	TO
Platoon30	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	−1	TO
Platoon42	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	−1	TO
SpaceRendezvous	OI	−1	−1	−1
	II	−1	−1	−1
	IIG	−1	−1	−1
	IIGB	−1	−1	−1
	OCT	−1	TO	TO

Support Function and Template Polyhedra Benchmark Results:

Benchmark	**Template Shape**	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$	$\delta = \mathbf{0.001}$
BouncingBall	OI	0.12s	0.18s	0.99s
	II	0.12s	0.18s	0.94s
	IIG	0.12s	0.21s	1.07s
	IIGB	0.12s	0.20s	1.08s
	OCT	0.12s	0.18s	0.82s
Building	OI	−1	−1	2.55s
	II	−1	−1	2.26s
	IIG	−1	−1	2.35s
	IIGB	−1	−1	2.07s
	OCT	−1	−1	2.05s
Navigation03	OI	39.36TO)	TO	TO
	II	36.29s	TO	TO
	IIG	44.59s	TO	TO
	IIGB	36.29s	TO	TO
	OCT	36.27s	TO	TO
Navigation04	OI	−1	−1	TO
	II	−1	−1	TO
	IIG	−1	−1	TO
	IIGB	−1	−1	TO
	OCT	−1	−1	TO
Navigation09	OI	−1	−1	−1
	II	−1	−1	TO
	IIG	−1	−1	TO
	IIGB	−1	−1	TO
	OCT	−1	−1	TO
Platoon30	OI	4.60s	6.43s	22.62
	II	4.60s	6.23s	21.97s
	IIG	4.61s	6.24s	18.16s
	IIGB	5.35s	6.84s	19.18s
	OCT	5.17s	6.22s	21.89s
Platoon42	OI	5.50s	6.33s	20.22
	II	4.59s	6.22s	18.40s
	IIG	4.60s	6.23s	18.15s
	IIGB	5.30s	7.33s	20.11s
	OCT	4.58s	6.22s	18.13s
SpaceRendezvous	OI	0.88s	13.98s	TO
	II	0.90s	11.06s	TO
	IIG	0.85s	12.86s	TO
	IIGB	0.85s	12.29s	TO
	OCT	0.84s	12.76s	TO

Further Optimization Results

Benchmark	**Template Shape**	$\delta = \mathbf{0.1}$	$\delta = \mathbf{0.01}$
navigation01	IIGB	6.24s	38.77s
	IIGB + FIX	1.19s	6.49s
	IIGB + FIX + NC	1.05s	5.66s
	OCT	110.44 s	TO
	OCT + FIX	42.93s	217.02 s
	OCT + FIX + NC	36.13s	194.20 s
navigation02	IIGB	9.48s	63.08s
	IIGB + FIX	1.72s	16.79s
	IIGB + FIX + NC	1.61s	13.34s
	OCT	125.93 s	TO
	OCT + FIX	63.26s	TO
	OCT + FIX + NC	46.83s	TO
navigation03	IIGB	5.79s	34.65s
	IIGB + FIX	1.61s	10.95s
	IIGB + FIX + NC	1.61s	10.82s
	OCT	73.95s	TO
	OCT + FIX	47.70s	244.34 s
	OCT + FIX + NC	41.89s	239.50 s
navigation04	IIGB	3.74s	23.13s
	IIGB + FIX	2.10s	11.52s
	IIGB + FIX + NC	1.85s	9.41s
	OCT	44.46s	TO
	OCT + FIX	44.93s	171.30 s
	OCT + FIX + NC	34.24s	162.69 s
navigation05	IIGB	−1	−1
	IIGB + FIX	−1	−1
	IIGB + FIX + NC	−1	−1
	OCT	−1	−1
	OCT + FIX	−1	−1
	OCT + FIX + NC	−1	−1
navigation06	IIGB	−1	40.20s
	IIGB + FIX	−1	17.56s
	IIGB + FIX + NC	−1	21.09s
	OCT	−1	TO
	OCT + FIX	−1	221.96 s
	OCT + FIX + NC	−1	225.84 s
navigation07	IIGB	17.34s	87.52s
	IIGB + FIX	4.47s	26.89s
	IIGB + FIX + NC	4.89s	27.01s
	OCT	194.86 s	TO
	OCT + FIX	110.42 s	TO
	OCT + FIX + NC	98.90s	TO
navigation08	IIGB	−1	89.47s
	IIGB + FIX	−1	30.82s
	IIGB + FIX + NC	−1	30.61s
	OCT	−1	TO
	OCT + FIX	−1	TO
	OCT + FIX + NC	−1	TO