Theory of
Hybrid Systems
Informatik 2

Master of Science Thesis

A Polytope Library for the Reachability Analysis of Hybrid Systems

Christopher Kugler

Supervisors:
Prof. Dr. Erika Ábrahám
Prof. Dr.-Ing. Stefan Kowalewski

Advisor:
M. Sc. Stefan Schupp

September 30, 2014

Abstract

Nowadays, hybrid systems are on the rise in fields such as control theory and therefore there is an urgent need for tools that are capable of verifying these systems. In context of a reachability analysis the reachable state set of a hybrid system is approximated, such that certain system properties can be examined on basis of the approximation.

This thesis covers a polytope centric implementation of such a reachability algorithm for the class of autonomous, linear hybrid systems and introduces the basic concepts beforehand. Furthermore, possible optimizations for the algorithm are explored. A special focus lies on the Minkowski sum operation, for which a recently proposed reverse search approach is implemented and evaluated.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Christopher Kugler
Aachen, den 30. September 2014

Acknowledgements

My gratitude goes to the whole group of Hybrid Systems at the university RWTH Aachen. In particular, I want to thank Prof. Dr. Erika Ábrahám for giving me the opportunity to write a thesis on such an interesting topic and for supporting me throughout the whole process. Furthermore, I thank Stefan Schupp as my advisor, who invested a huge amount of hours and provided an excellent introduction into the development environment as well as the theoretical background. Also, I appreciate the services of Prof. Dr. Stefan Kowalewski as my second supervisor. At last, I deeply thank my family and friends for their support and encouragement, which helped overcome even the most work-intensive phases.

Contents

Chapter 1

Introduction

This thesis resides in the field of system modeling and verification. Here, the general notion is to abstractly model a real-world system and then perform verification with respect to some property on this abstraction. This procedure may for instance be used for validating complex systems, which are often found in context of control theory and physical processes. Imagine an *air traffic control (ATC) system*: Its purpose is to manage the airplane traffic and provide the pilots with crucial and safety-relevant information. One of the ATC's most important tasks involves collision avoidance for airplanes, for which complex computations based on several parameters have to be conducted. To decide whether a collision may occur, it is then a question of determining if a certain, marked as dangerous, system state may be reached depending on the input data that is steadily generated by the airplanes. Therefore, by performing a *reachability analysis* on an abstraction of the ATC, it is possible to deduce whether a collision is impending or not.

In context of this thesis the focus will be on so-called *hybrid systems*, which exhibit both discrete as well as continuous characteristics. To provide an adequate abstraction, such a system is modeled by a certain class of automata: *hybrid automata*.

Essentially, by performing a reachability computation it would be possible to derive statements about whether the system satisfies a certain property, may that be safety or some other characteristic according to the specified system requirements. However, the reachability problem for hybrid systems is in general undecidable [LG09]. As such, we have to resort to over-approximating the system states and if our over-approximation is considered safe then the system itself can be assumed to be so too.

Such an over-approximation is done on the basis of the hybrid automaton (i.e. the abstraction that describes the behaviour of the hybrid system). In general there are various approaches on how to perform the reachability computation, each with distinct computational characteristics. Therefore, the aim of this thesis is to provide an efficient and accurate implementation of a reachability algorithm for hybrid systems, which is ought to be usable as part of a library.

Outline of the Thesis

Chapter 2 first provides background knowledge that is required in the context of reachability analysis. Specifically, Section 2.1 introduces the class of hybrid automata, while Section 2.2 continues with existing state set representations and operations that need to be performed on these sets.

Chapter 3 introduces the general reachability algorithm, where reachability in context of locations (Section 3.2) is considered in detail, followed by transitions (Section 3.3). In Chapter 4 possibilities for optimization are investigated. The focus is set on the Minkowski sum operation, for which an algorithm that has been proposed in recent literature is introduced.

Chapter 5 provides implementation details for both the Minkowski sum and the reachability algorithm as well as respective evaluation results. Lastly, Chapter 6 gives insight into related work before Chapter 7 concludes the thesis.

Chapter 2

Background

Before presenting the theoretical details of the reachability algorithm in Chapter 3, first some foundations and terminology shall be established in the following.

2.1 Hybrid Automata

Hybrid automata provide a formal model for hybrid systems. In principle, a hybrid automaton may be seen as an extension of ordinary discrete automata. Whereas discrete automata aim to model discrete behaviour, i.e. digital processes, a hybrid automaton is also capable of describing continuous behavior, i.e. analog processes.

A hybrid automaton is a tuple $\mathcal{H} = (Loc, Var, Lab, Inv, Act, Trans, Init)$ [LG09], where:

- *Loc* denotes a finite set of locations.
- *Var* is a finite set of variables with real domain.
- *Lab* is a finite set of synchronization labels, where $\tau \in Lab$ denotes the empty label. Given a *network* of hybrid automata, such labels may be used to synchronize discrete steps between multiple automata.
- $Inv \subseteq Loc \times \mathbb{R}^d$ is a set of invariants, where $d = |Var|$. An invariant $I_l \in Inv$ is mapped to location $l \in Loc$ and restricts the values of the variables in this location.
- $Act \subseteq Loc \times \mathbb{R}^d \times \mathbb{R}^d$ denotes a set of activities. Again, $a_l \in Act$ is specific to a location $l \in Loc$ and a_l holds a differential equation describing the change of a variable over time.
- $Trans \subseteq Loc \times Lab \times 2^{\mathbb{R}^d \times \mathbb{R}^d} \times Loc$ is the set of transitions of the automaton. A transition $t \in Trans$ is of the form $t = (l, a, \mu, l')$, where:
  - $l \in Loc$ is the source location and $l' \in Loc$ is the target location
  - $a \in Lab$ is the synchronization label for t

- μ is the *reset map*: For any two valuations v,v', if $(v,v') \in \mu$ and $(l',v') \in \mathit{Inv}$, then the automaton may jump from state (l,v) to (l',v'). A reset map μ consists of a *guard* and a *reset* relation. The transition is only *enabled* if its guard is fulfilled by v and if the transition is taken then the reset is applied on the variables of the automaton, changing the current variable valuation to v'.

- $\mathit{Init} \subseteq \mathit{Loc} \times \mathbb{R}^d$ is the set of initial states, specifying initial locations and their initial valuations.

A *state* $s \in \mathit{Loc} \times \mathbb{R}^d$ of a hybrid automaton is described by a location $l \in \mathit{Loc}$ and a valuation over Var, which assigns a value $v_i \in \mathbb{R}$ to each variable $x_i \in \mathit{Var}$, $i \in \{1,..,d\}$.

Figure 2.1 shows the graphical representation of a hybrid automaton $\mathcal{H} = (\{l_1,l_2\},\{x,y\},\{\tau,a\}\{(l_1,x \leq 1),(l_2,\mathit{true})\},\{a_{l_1},a_{l_2}\},\{t_1 : (l_1,a,x := 0,l_2), t_2 : (l_2,\tau,x \geq 15 \rightarrow x := 0,l_1)\},\mathit{Init})$.

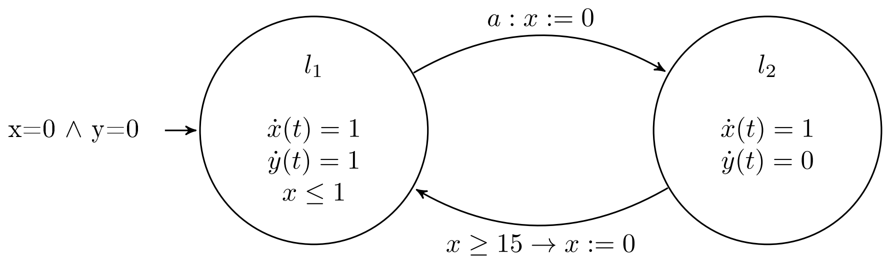


Figure 2.1: Exemplary hybrid automaton $\mathcal{H}$.

Here, the automaton consists of two locations l_1,$l_2 \in \mathit{Loc}$ and two variables x,$y \in \mathit{Var}$. The initial state is described by $\mathit{Init} = \{(l_1,(0,0)^T)\}$, where both variables are initially set to zero. The activities are given by $a_{l_1} : \{\dot{x}(t) = 1, \dot{y}(t) = 1\}$ and $a_{l_2} : \{\dot{x}(t) = 1, \dot{y}(t) = 0\}$. Also, there is an invariant $I_{l_1} \in \mathit{Inv}$ for location l_1, which limits the valuation of x to less than or equal to 1. The invariant of l_2 is *true*. Furthermore, two transitions $t_1 = (l_1,a,\mu_1,l_2)$ and $t_2 = (l_2,\tau,\mu_2,l_1)$ are specified. Formally, $\mu_1 = \{(v,v') \in V^2 \mid v'(x) = 0\}$ and $\mu_2 = \{(v,v') \in V^2 \mid v(x) \geq 15 \wedge v'(x) = 0\}$, signifying that t_1 has no guard but performs a reset of x to 0, whereas t_2 performs the same reset but is only enabled if the guard $x \geq 15$ is fulfilled.

For the formal semantics of hybrid automata we refer to [HKPV95].

2.1.1 Linear Hybrid Automata

In this thesis the focus will be on *linear hybrid automata*, meaning that activities $a_l \in \mathit{Act}$ may be described by linear ordinary differential equations (linear ODE) [LG09]:

$$\dot{x}(t) = Ax(t) \tag{2.1}$$

where A is a $d \times d$ matrix with d specifying the number of variables in the automaton and respectively the dimension of the model. In general, a linear hybrid system may also contain a constant part b, i.e. $\dot{x}(t) = Ax(t) + b$. However it is possible to rewrite this notation to fit Equation 2.1 by encoding b into a new matrix A'. The details of this procedure will be explained in Chapter 5.

For linear ODEs, solutions may be computed using the following equation [LG09]:

$$x(t) = e^{tA}x_0 \tag{2.2}$$

Here, $x(0) = x_0$ is the initial value of x and e^{tA} denotes the *matrix exponential*, which is defined as [MVL03]:

$$e^A = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{2.3}$$

Equation 2.2 is relevant for the reachability analysis procedure presented in Chapter 3.

2.1.2 Autonomous Systems

In general the systems to be analyzed may have an additional external *input* $u(t)$, such that the ODE is of form [LG09]:

$$\dot{x}(t) = Ax(t) + u(t) \tag{2.4}$$

with solution:

$$x(t) = e^{tA}x_0 + \int_0^t e^{(t-s)A}u(s)ds \tag{2.5}$$

We call these systems *non-autonomous*. However, in this thesis the focus will be on *autonomous systems*, i.e. ODEs that satisfy the notation given in 2.1. An extension of the reachability algorithm for non-autonomous systems is part of future work.

2.2 Reachability Analysis

As mentioned in the introduction, the reachability problem for hybrid automata is generally undecidable [HKPV95]. Therefore, instead of computing the exact set of reachable states, an alternative is to compute an over-approximation of this set. Such an over-approximation may be computed using one of many possible set representations, some of which will be introduced in the following. With few exceptions, a set representation is a geometric object that describes the possible variable valuations for a system state. Figure 2.2 illustrates the general notion of this approach, with the exact behaviour being depicted on the left and the approximation by rectangles on the right.

The basic idea to retrieve the set of reachable states is a fixed point calculation. Starting from the initial states *Init*, the general procedure is to take both continuous (i.e. let time elapse) and discrete steps (i.e. take a transition in the automaton) until no new states can be explored. However, since some hybrid systems may have an infinite state space, for the computation to finish it has to be limited by either a time or discrete step boundary (or a combination of both).

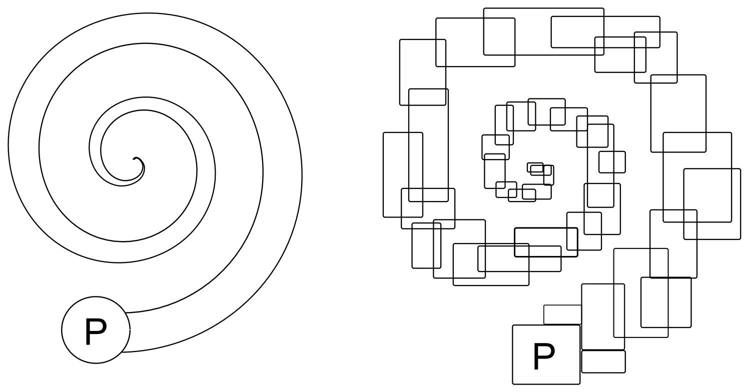


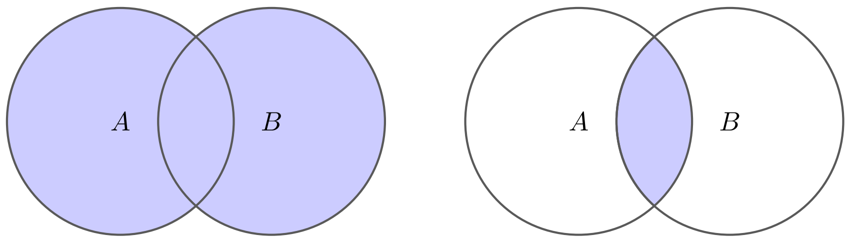
Figure 2.2: Exemplary reachability approximation.

2.2.1 Operations

In the following, the set operations that are needed to perform the reachability computation are introduced in detail:

- Computing the union $\cup$ of two sets or the convex hull CH over such a union
- Intersection $\cap$ of either two sets or one set with a hyperplane
- Linear transformation of a set (by multiplication with a matrix A)
- Computing the Minkowski sum of two sets

The union of two sets A,B is the collection of all those set elements which are in either A or B: $A \cup B = \{x \mid x \in A \vee x \in B\}$. In contrast, the intersection of two sets denotes those elements that are shared between both sets: $A \cap B = \{x \mid x \in A \wedge x \in B\}$. Both of these operations are graphically illustrated in Figure 2.3 for two sets in $\mathbb{R}^2$.



(a) $A \cup B$. (b) $A \cap B$.

Figure 2.3: Union and intersection of two sets.

A linear transformation T is a mapping from one vector space V to another vector space W [Str03]. By definition, this mapping preserves both addition and multiplication with a scalar:

$$\begin{aligned} T(v_1 + v_2) &= T(v_1) + T(v_2) \\ T(\alpha v) &= \alpha T(v) \end{aligned} \tag{2.6}$$

where v, v_1, v_2 are vectors in V and α is a scalar. Furthermore, lines are always mapped to lines or to zero. The matrix multiplication $T(v) = Av$ for a $n \times m$ matrix A is an example of a linear transformation from $V = \mathbb{R}^m$ to $W = \mathbb{R}^n$.

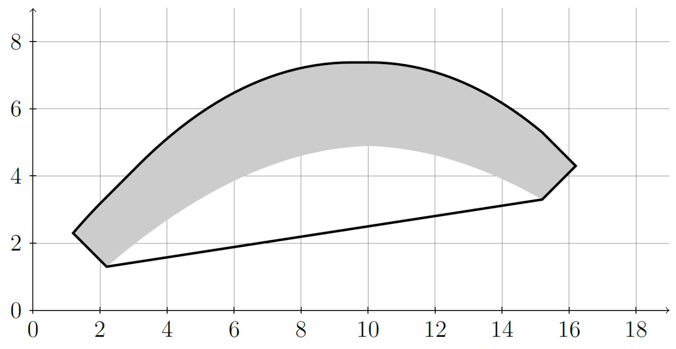


Figure 2.4: An illustration of the convex hull. [LG09]

The convex hull $CH(\mathcal{P})$ of a non-convex set $\mathcal{P}$ refers to the smallest convex set that contains P. For instance, Figure 2.4 illustrates the convex hull for the given set.

The Minkowski sum of two sets $\mathcal{A}$ and $\mathcal{B}$ is formally defined as follows [LG09]:

$$\mathcal{A} \oplus \mathcal{B} = \{a + b \mid a \in \mathcal{A}, b \in \mathcal{B}\} \tag{2.7}$$

Essentially, every element in set $\mathcal{A}$ is added up with every element in set $\mathcal{B}$, yielding a set of sums. Graphically, such a computation describes sliding one set along the edges of the other set and taking the union of all subsets that have been observed while doing so (ref. Figure 2.5).

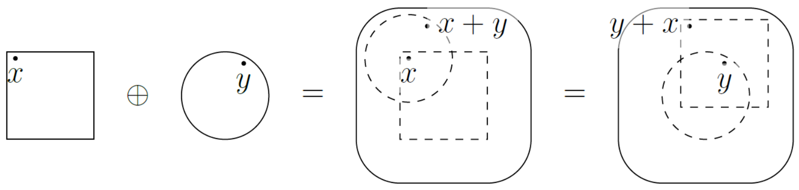


Figure 2.5: An illustration of the Minkowski sum. [LG09]

2.2.2 Representations

This section briefly introduces four possible geometric set representations. The common characteristic among all representations is their convexity, meaning that for any two points a and b of a convex set $P \subseteq \mathbb{R}^d$, every point on the line segment between a and b is also contained in the set:

$$\forall a,b \in P.\ \forall x \in [0,1]^d \subseteq \mathbb{R}^d : a + x(b - a) \in P \tag{2.8}$$

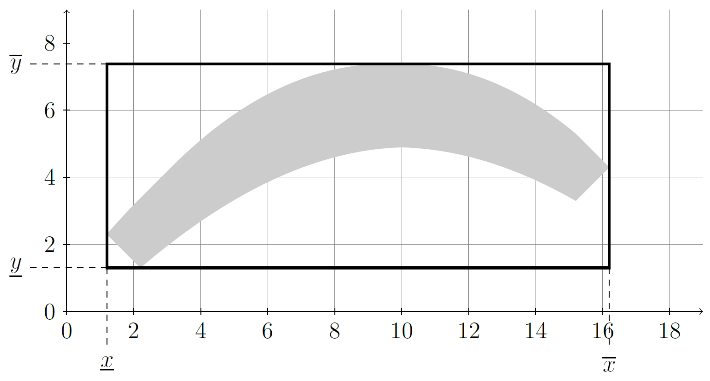


Figure 2.6: Approximating a set by a (hyper-)rectangle. [LG09]

Hyper-rectangle. A hyper-rectangle $\mathcal{R}$ is the generalization of a rectangle to an arbitrary dimension d, which can be mathematically described as a product of intervals [LG09]:

$$\mathcal{R} = [\underline{x_1}, \overline{x_1}] \times \ldots \times [\underline{x_d}, \overline{x_d}] \tag{2.9}$$

As Figure 2.6 shows, approximating a set with a hyper-rectangle is rather inaccurate, however it can be computed and stored very efficiently.

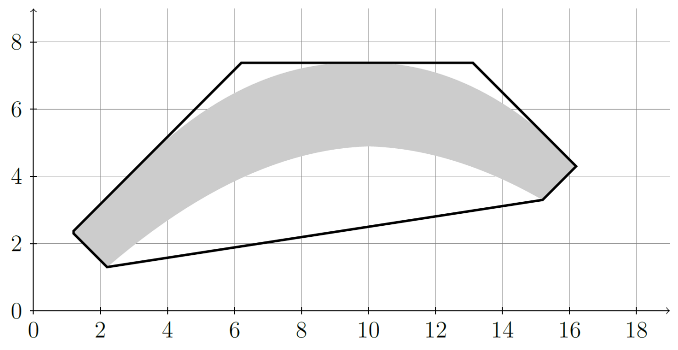


Figure 2.7: Approximating a set by a polytope. [LG09]

Polytope. A d-dimensional halfspace is a set $h = \{x \in \mathbb{R}^d \mid c^T x \leq z\}$, where $c \in \mathbb{R}^d$ is the normal of h and $z \in \mathbb{R}$ is the offset. A polyhedron[1] $\mathcal{P}$ is the intersection of halfspaces from a finite set $\mathcal{H}$:

$$\mathcal{P} = \bigcap_{h \in \mathcal{H}} h \tag{2.10}$$

A polytope is a bounded polyhedron and may be defined in two different ways: Either by the set $\mathcal{V}$ of its vertices ($\mathcal{V}$-representation), or by encoding

[1]In this thesis we consider only convex polyhedra.

the halfspaces $\mathcal{H} = \{c_i^T x \leq z_i \mid i = 1,...,m\}$ as $\big((c_1,...,c_m)^T, (z_1,...,z_m)^T\big) \in \mathbb{R}^{m \times d} \times \mathbb{R}^d$ ($\mathcal{H}$-representation).

Each of those representations has their own benefits and drawbacks with respect to computational efficiency when performing certain operations on polytopes [Zie95]. Therefore it is common to make use of both representations, however the translation from one representation to the other may be exponential in the state space dimension [Zie95]. Figure 2.7 provides an example of how a set may be approximated using a polytope.

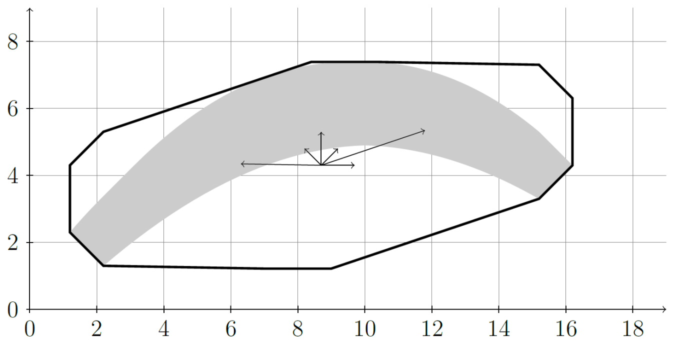


Figure 2.8: Approximating a set by a zonotope. [LG09]

Zonotope. A zonotope $\mathcal{Z}$ in $\mathbb{R}^d$ can be defined as the finite Minkowski sum of line segments, also called generators of $\mathcal{Z}$, shifted by a vector $c \in \mathbb{R}^d$ [LG09]:

$$\mathcal{Z} = c + [-1,1]v_1 \oplus ... \oplus [-1,1]v_k \tag{2.11}$$

where $v_i, i \in \{1,..,k\}$ is a d-dimensional vector. As illustrated in Figure 2.8, zonotopes always have a center of symmetry and form a subclass of polytopes.

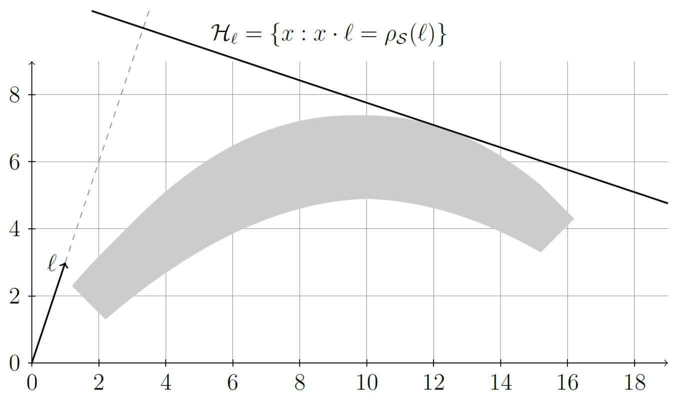

$\mathcal{H}_\ell = \{x : x \cdot \ell = \rho_{\mathcal{S}}(\ell)\}$
ℓ
0
2
4
6
8
0
2
4
6
8
10
12
14
16
18

Figure 2.9: A hyperplane given by a support function. [LG09]

Support Function. Support functions are different from the other set representations because here, convex sets are represented by a function instead of a set of parameters. Formally, the support function $p_s : \mathbb{R}^d \to \mathbb{R} \cup \{-\infty, \infty\}$ of a set $\mathcal{S}$ is defined as [LG09]:

$$p_s(l) = \sup_{x \in \mathcal{S}} x \cdot l \tag{2.12}$$

where $l \in \mathbb{R}^d$. In essence, a support function maps any direction l to a hyperplane $\mathcal{H}_l$ which is orthogonal to l, such that $\mathcal{H}_l$ contains the whole set and its surface is tangential to the set that is to be approximated. Figure 2.9 pictures one such hyperplane for a given direction l. Depending on the amount of directions l_i considered, the set is approximated more accurate with increasing i, culminating in the intersection of the $\mathcal{H}_{l_i}$.

In context of this thesis reachable state sets are approximated by polytopes, which allow for a lot of flexibility.

Table 2.10 shows the complexity of the previously introduced operations for different set representations. Here, "$+$" indicates low complexity, "$*$" low complexity for most instances and "$-$" high complexity. An empty field signifies that the operation is not feasible using the respective representation.

	$A\cdot$	$\cdot \oplus \cdot$	$\mathrm{CH}\,(\cdot \cup \cdot)$	$\cdot \cap \mathcal{H}$
Boxes		$+$		
$\mathcal{H}$-Polytopes	$*$	$-$	$-$	$+$
$\mathcal{V}$-Polytopes	$+$	$-$	$+$	$-$
Zonotopes	$+$	$+$		
Support Functions	$+$	$+$	$+$	$-$

Figure 2.10: Complexity of operations depending on representation. [LG09]

The computation of the Minkowski sum stands out as being computationally hard for both polytope representations. As pointed out in [Wei07], it is difficult to find an efficient algorithm and the resulting representation is often dramatically increased in size. The fundamental problem of a brute force computation of the Minkowski sum, that is to add up every element in one set with every element in the other set (based on a $\mathcal{V}$-representation), is the fact that a significant amount of computed vertices are inner vertices of the resulting polytope. For both high dimensions and a high amount of vertices in each summand this leads to a very high computational effort, which is further increased by having to compute the convex hull of the result to recover the $\mathcal{V}$-representation and thus to reduce the representation size.

Consequently, a major focus of this thesis is the optimization of the Minkowski sum operation in context of $\mathcal{V}$-Polytopes, with the aim of significantly improving the performance of the reachability computation for complex hybrid systems.

Chapter 3

Reachability Analysis

Recalling the purpose of reachability analysis as a whole, it is to identify the set of reachable states of a given hybrid system under consideration of some time or discrete step bound. As previously stated, the basic idea is to approximate the exact reachability result using an appropriate state set representation, in this case polytopes. Each computed polytope represents the set of possible valuations the system can reach at some point in time. Based on this over-approximation it is possible to assert certain system properties, such as whether the system is safe with respect to some pre-defined set of malicious states, also represented using the chosen representation.

In general, considering only continuous progress within one location of a hybrid automaton (i.e. letting time advance up to a fixed time point), the reachability result as a whole may be considered to be a *flowpipe* (ref. Figure 3.1). Given a discretization factor f, a flowpipe may be divided into f connected parts, called *flowpipe-segments*, where the union over all segments yields the flowpipe itself again. Each of these segments is approximated by a polytope, and consequently the whole flowpipe is formed by a set of polytopes.

Moving on to discrete transitions between the locations of a hybrid automaton, the previously introduced *reset map* yields a new variable valuation in the target location. Given this valuation, a flowpipe for this target may be computed as well, such that the reachable state set of a construct consisting of two locations and a transition in between may simply be considered to be the union of both flowpipes.

3.1 General Reachability Algorithm

In Listing 1 the general reachability algorithm is described abstractly [Ábr12].

The fundamental idea of this algorithm is to start with the set of initial states *Init* as given by the hybrid automaton (line 1 of Listing 1) and to compute a first flowpipe approximation based on this input set in context of the function `Reach(..)` (line 5). After that, `Reach(..)` performs one discrete step in which every outgoing transition of the initial location(s) is taken into consideration: By intersecting the *guard* of a transition's reset map with each flowpipe segment individually, it is possible to identify non-empty intersection parts. On these parts, which are itself polytopes, the *reset* relation of

```
Input   : Set $Init$ of initial states
Algorithm:
1       $R_{new} := Init$;
2       $R := \emptyset$;
3       while ($R_{new} \neq \emptyset$) do
4           $R := R \cup R_{new}$;
5           $R_{new} :=$ Reach $(R_{new}) \setminus R$;
        end
Output: Set $R$ of reachable states
```

Algorithm 1: General reachability algorithm.

the reset map is then applied, yielding a new valuation in the target location. Continuing this procedure for each location that is observed while performing the reachability computation, the final result is a set of flowpipes for each location (as one location may have multiple incoming transitions). The union of all these location-specific flowpipes (line 4) describes an over-approximation for the reachable state set of the whole hybrid automaton, where a state is defined as a tuple consisting of a location and the respective variable valuation.

In the following the concept of reachability in context of locations and transitions is explained in detail.

3.2 Reachability in Context of Locations

Given an initial valuation, the reachability of a single location may be described by computing a bounded over-approximation of the exact flowpipe. The underlying idea behind such a flowpipe is to let time elapse within one location and observe the possible valuations of all variables (which spans the *state space*). Specifically, the variables evolve according to the location's activity $a_l \in Act$, which may be interpreted as a differential equation that is dependent on a time variable t. Based on the computed data it is possible to assert whether a certain (location-specific) state may be reached during system execution or not. An exemplary flowpipe approximation is depicted in Figure 3.1.

3.2.1 Flowpipe Computation

There are various approaches in literature for computing an over-approximation of a flowpipe for one location of a hybrid automaton, each with different characteristics regarding both efficiency as well as accuracy. A brief overview for these approaches is given in Chapter 6, while in the following the approach that has been implemented in context of this thesis will be explained in detail.

Recalling Equations 2.1 and 2.2, activities $a_l \in Act$ are considered to be linear ODEs:

$$\dot{x}(t) = Ax(t)$$

with solutions being of the form:

$$x(t) = e^{tA}x_0$$

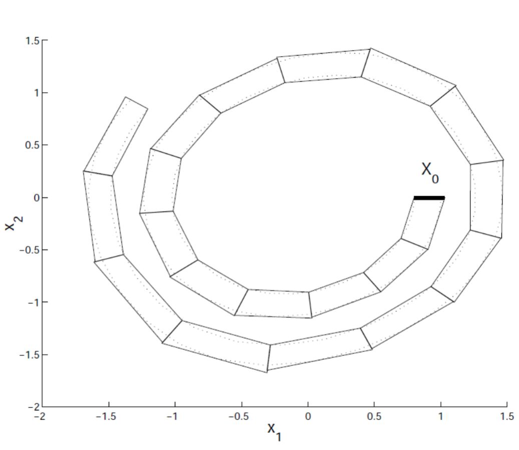


Figure 3.1: Exemplary approximation of a flowpipe, where X_0 denotes the initial set. The flowpipe is divided into twenty segments, each of which is approximated by a single polytope. [CK98]

The matrix A is referred to as the *activity matrix* of a location, storing the coefficients of the ODE. Applied to any set $\mathcal{S}$, the reachable variable valuations $\mathcal{R}_\delta(\mathcal{S})$ in location l at time $t = \delta$ are given by the expression $e^{\delta A}\mathcal{S}$, which describes a linear transformation of set $\mathcal{S}$ with matrix $e^{\delta A}$ [LG09]. Consequently, the sequence of reachable sets Ω_0, ... ,Ω_T that is to be computed may be defined by a recurrence relation:

$$\Omega_{i+1} = e^{\delta A}\Omega_i \tag{3.1}$$

Here, each Ω_i represents an approximation of one flowpipe-segment. δ refers to the (static) timestep size that is used for the approximation, i.e. segment Ω_0 would approximate the time interval $[0,\delta]$, Ω_1 the time interval $[\delta, 2\delta]$, continuing up to Ω_T, where $T = n\delta$ signifies the upper time bound. As a result of this approximation scheme, a smaller δ results in a more granular segmentation of the flowpipe, yielding higher accuracy at the cost of greater computational effort.

Furthermore, two things are noteworthy concerning Equation 3.1: First, since δ is a constant it is sufficient to compute $e^{\delta A}$ only once per location. As $e^{\delta A}$ describes the computationally very expensive matrix exponential (ref. Equation 2.3), this is a great benefit of any approximation approach that is based on recurrence relations. Even more so, depending on the implementation of the matrix exponential, its computation might not even terminate in reasonable time for huge δ [MVL03].

Secondly, the upper time bound T defines the last flowpipe segment that is considered in the reachability analysis of one location l. This timebound might be an artificial bound passed along to the algorithm as a parameter to guarantee termination (i.e. if the reachable state space is infinite). However, the variable valuations and as such also the flowpipe as a whole is primarily bounded by the location-specific invariant $I_l \in \mathit{Inv}$. Therefore, while computing the approximation for each flowpipe segment according to the recurrence relation, in each step it has to be checked whether the next segment still satisfies the bounds given by the invariant. This is done in context of a containment test, since the invariant may also be represented as either a polytope or a hyperplane.

Given the recurrence relation, the remaining part is to approximate the initial flowpipe segment Ω_0, which is used as a basis for all approximations of later segments.

Approximating Ω_0

The approximation of Ω_0 depends on the initial valuation in a location. We assume any state set description to be given by a polytope. For a starting point of the algorithm it is reasonable to consider the set of initial states *Init*, which defines one or multiple starting locations with individual variable valuations. As it is indifferent for the general procedure whether there is more than one starting location, in the following it will be assumed that $|Init| = 1$ for purposes of simplicity. Let $Init = \{(l,X_0)\}$, where X_0 refers to the initial variable valuations.

Ω_0 may then be approximated according to the following equation [LG09]:

$$\Omega_0 = CH(R_0(X_0) \cup R_\delta(X_0)) \oplus B(\alpha_\delta) \tag{3.2}$$

where:

- $R_i(X_0)$ refers to the manifestation of X_0 at time i with respect to the previous recurrence relation. For instance, $R_0(X_0) = e^{0A}X_0 = X_0$ and $R_\delta(X_0) = e^{\delta A}X_0$.
- $\cup$, $CH(\cdot)$ and $\oplus$ refer to the set operations union, convex hull and Minkowski sum respectively.
- $B(\alpha_\delta)$ describes the ball of appropriate dimension with radius α_δ. Note that the shape of the ball entirely depends on the chosen norm, which for our purpose is induced by the computation of α_δ.
- α_δ is an upper bound for the so-called *Hausdorff distance* between the approximation and the exact flowpipe.

The Hausdorff distance $d_H(\mathcal{A},\mathcal{B})$ for two sets $\mathcal{A}$ and $\mathcal{B}$ is formally defined as follows [LG09]:

$$d_H(\mathcal{A},\mathcal{B}) = \max\left\{\sup_{a\in\mathcal{A}}\inf_{b\in\mathcal{B}}\|a-b\|, \sup_{b\in\mathcal{B}}\inf_{a\in\mathcal{A}}\|a-b\|\right\}$$

In essence, d_H is the greatest of all distances from a point in one set to the closest point in the other set. Figure 3.2 illustrates the meaning of the Hausdorff distance for two exemplary sets.

In the following, the procedure of approximating Ω_0 is examined in detail. An intermediate goal of the approximation is shown in Figure 3.3a. Here, the dotted line represents the exact flowpipe, while the bold polytope denotes a first approximation. This approximation is computed by the first part of Equation 3.2: $CH(R_0(X_0) \cup R_\delta(X_0))$. Specifically, $R_0(X_0) = X_0$ refers to the initial valuations given by (l,X_0). According to the recurrence relation a linear transformation is applied to X_0 to compute a further valuation that is also part of the exact flowpipe: $R_\delta(X_0) = e^{\delta A}X_0$. These two valuation sets then serve as a basis for the approximation, and by computing the convex hull over the union of

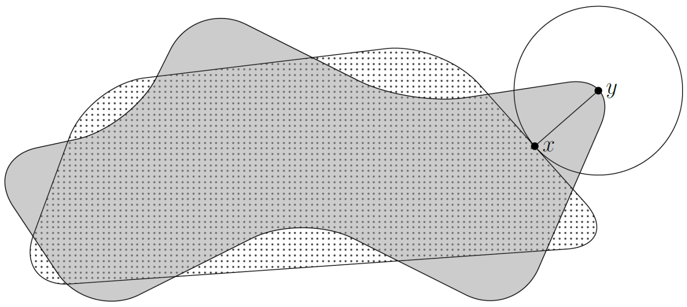
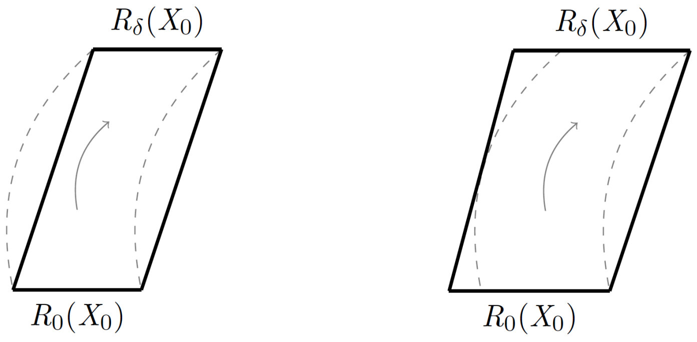


Figure 3.2: Illustration of the Hausdorff distance. [LG09]

both sets the set in Figure 3.3a is obtained. However, this first approximation is not sufficient, as there is still some part of the exact flowpipe which is not contained within the approximation. Having an over-approximation is important for being able to assert certain system properties, e.g. safety, because in contrast to an under-approximation a statement about the over-approximated state set still holds for the exact flowpipe, which is not computable due to the undecidability of the problem [HKPV95].



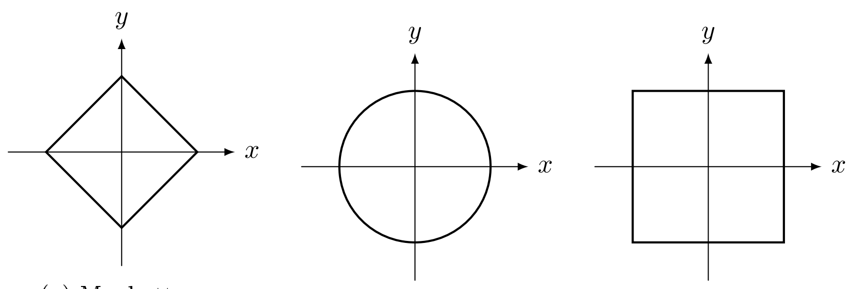
(a) Intermediate approximation result. (b) Final (bloated) approximation result.

Figure 3.3: Approximation process for one flowpipe segment.

To receive an over-approximation, the hull of the intermediate result is *bloated*. To do so, the Minkowski sum of the intermediate result and the ball $B(\alpha_\delta)$ is computed. While there are multiple ways to approximate the Hausdorff distance between two sets, the following approach based on the infinity norm $\|\cdot\|_\infty$ is implemented in context of this thesis [Gir05]:

$$\alpha_\delta = (e^{\delta||A||} - 1 - \delta||A||) \max_{x \in X_0} ||x|| \tag{3.3}$$

Consequently, the ball $B(\alpha_\delta)$ is also defined with respect to the infinity norm. This is very important, as the ball then has the shape of a *hypercube* as Figure 3.4 visualizes. Since any hypercupe is also a polytope, we preserve the polytope representation through the application of the Minkowski sum.



(a) Manhatten norm: $\|x\|_1$. (b) Euclidean norm: $\|x\|_2$. (c) Infinity norm: $\|x\|_\infty$.

Figure 3.4: Unit ball in $\mathbb{R}^2$ for different norms.

One possible over-approximation that is a result of bloating the hull is shown in Figure 3.3b. Note that the depicted case is an ideal over-approximation, where each edge of the polytope touches one part of the exact flowpipe. In general the hull will be bloated to a greater effect, meaning that there might be a considerable over-approximation in all directions. As one summand of the Minkowski sum is a ball, every dimension will be bloated accordingly, even if not necessary.

Having computed an approximation Ω_0 for the first flowpipe segment, applying a linear transformation on this segment according to the recurrence relation 3.1 yields the next segment. This way, the whole flowpipe of a location may be approximated using a sequence of polytopes, describing the set of reachable states when letting time elapse in one location.

3.3 Reachability in Context of Transitions

Having considered reachability in the context of single locations, the next step is to involve discrete transitions between multiple locations and examine their impact on the reachable state space. Let $t = (l, a, \mu, l')$ be a transition where the flowpipe approximation for location l has already been computed. The transition t is enabled when the guard defined by the reset map μ is satisfied for a given variable valuation. A guard may be stored as either a hyperplane or a polytope and therefore the intersection between the guard and each approximated flowpipe segment can be computed in context of the reachability algorithm. If the intersection is not empty, the transition may be taken from the state space defined by the intersection. In general the concept of a transition involves nondeterministic behaviour, as it is not known whether a transition is taken even if its guard is enabled. However, since the intent is to over-approximate the reachable state set, one valid approach is to just compute all possibilities. Consequently, the reachability algorithm applies the reset of μ to every non-empty intersection individually, yielding a set of new valuations in the target location. Here, a reset is assumed to primarily be a linear transformation, followed by an optional translation in any dimension.

Figure 3.5 illustrates the intersection of a hyperplanar guard g (indicated by the blue line, where everything above fulfills the guard) with a given flowpipe approximation.

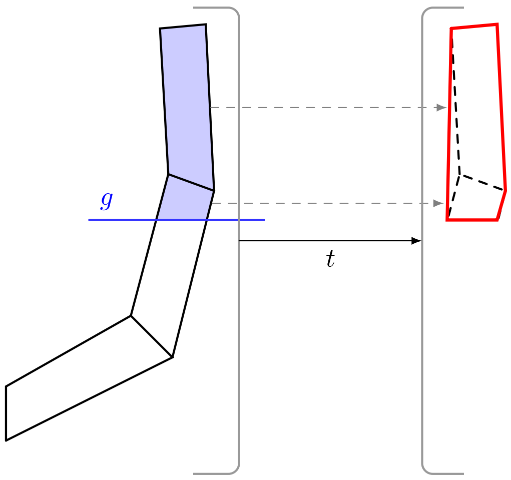


Figure 3.5: Intersection of a hyperplanar guard g with a flowpipe. The reset map of transition t is applied, yielding a new initial state set in the target location.

After applying the reset of transition t, it is necessary to first unite the new valuation sets in the target location. As the union of connected sets is generally not a convex polytope, we also need to compute the convex hull as indicated in Figure 3.5 by the solid red line. This convex hull is a valid X_0 for the target location, which may be used to approximate the first flowpipe segment Ω_0 and therefore also the reachable state set in the given context. However, there may be another, different X_0 when taking a transition to l' from any location other than l as shown in Figure 3.6. Considering this possibility a location may produce multiple flowpipe approximations, which are all relevant for the reachability analysis procedure.

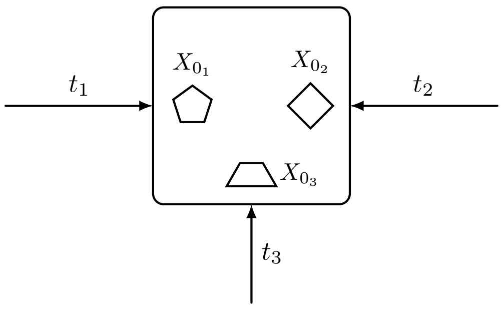


Figure 3.6: Each incoming transition contributes to one initial state set X_{0_i}.

Having seen how the reachability algorithm operates in detail, we deal with some limitations of the presented approach in the next section.

3.4 Limitations

While there are a lot of computational benefits to the reachability computation approach presented in this chapter, its accuracy might be improved in context of future work.

For instance, the approximation error of the first segment Ω_0 is carried over into the computation of later segments, as per recurrence relation only a linear transformation is applied on the preceding segment. This might cause a large error accumulation especially for the last parts of the flowpipe. An alternative would be to recompute the approximation of a flowpipe segment after e.g. a fixed amount of segments have been observed. This implies that a further exact segment of the flowpipe X_p is computable, which could then be used to generate a better flowpipe approximation Ω_p. However, it holds that $X_p = e^{pA}X_0$, but in contrast to the usual step size δ, p is much greater and only increases each time a new exact segment shall be computed. As outlined in [CK98], computing the matrix exponential for big p may prove difficult if not even impossible.

A further factor that could theoretically lessen the size of the over-approximation is the handling of the multiple valuations that result of a transition's reset. Instead of uniting and then computing the convex hull, which includes "irrelevant" state space, one approach would be to just handle the result of each segment reset individually. One transition would then cause the emergence of multiple flowpipes in the target location. However, if a system is of any considerable complexity such an approach would probably cause a state explosion, rendering the computation infeasible.

Chapter 4

Optimization

In context of the reachability analysis approach that has been presented in Chapter 3, there are many opportunities for optimizing different parts of the procedure. A major factor that influences the computational complexity of the algorithm is the efficiency of the operations that are applied on the set representations (i.e. on polytopes). As previously outlined in Section 2.2.2, the Minkowski sum is explicitly hard to compute independently of whether a $\mathcal{V}$- or $\mathcal{H}$-representation is used. In the following a different approach for the computation of the Minkowski sum based on the findings by Komei Fukudua [Fuk04] shall be examined, which has also been implemented as part of this thesis.

4.1 Minkowski Sum

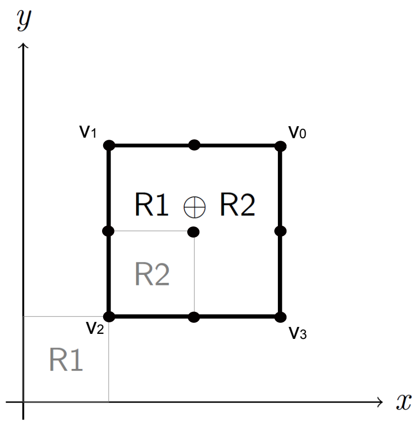


Figure 4.1: Computed vertices of a brute force Minkowski sum approach.

For complex hybrid systems that cause the approximative polytopes to be of high dimension and to have a high amount of vertices, the standard *brute force* Minkowski sum computation may cause a huge computational overhead. As every vertex of one summand is just aggregated with every vertex of the other summand, the result is exponential in its output size [Wei07] [Fuk04]. Even worse, in order to reduce the $\mathcal{V}$-representation, the convex hull has to be computed over the returned set of points (as some points may not be vertices of the sum polytope, but inner points instead), which may be computationally very expensive by itself. On the other hand, if the representation is not reduced, its size increases significantly.

Figure 4.1 illustrates the brute force computation for two polytopes R_1 and R_2 in $\mathbb{R}^2$. We observe that in total nine unique points have been computed, although only $v_0, ..., v_3$ are vertices and as such belong to the $\mathcal{V}$-representation of the sum polytope $R_1 \oplus R_2$. Indeed, some computed points are redundant and consequently the brute force approach returns $4 \cdot 4 = 16$ instead of the required four points. While duplicates may be recognized during the computation, to get rid of any inner points, the convex hull has to be computed on the given set of points.

An alternative approach proposed by Fukuda [Fuk04] is tailored for the Minkowski addition of k polytopes $P_i, i \in \{1,...k\}$, with dimension d and assumes that every summand is a $\mathcal{V}$-polytope. The general idea is to perform a reverse search on the directed *spanning tree* of the sum polytope $P = P_1 + ... + P_k$, which is derived from the neighbor relations of the summands P_i. In principle the algorithm traverses all possible edge directions in P, iteratively enumerating the vertices as soon as they are identified.

4.1.1 Reverse Search

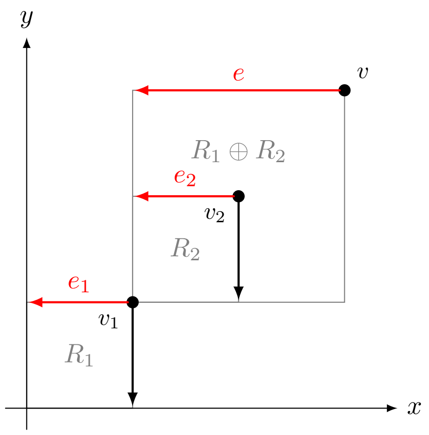


Figure 4.2: Vertex and edge decomposition in $R_1 \oplus R_2$.

Fukuda's algorithm makes use of several properties of the Minkowski sum. The two most important are given by the following propositions [Fuk04] [GS93]:

Proposition 1: *Let P_1,..,P_k be polytopes in $\mathbb{R}^d$ and let $P = P_1 + ... + P_k$ be the Minkowski sum result. A vector $v \in P$ is a vertex of P if and only if $v = v_1 + ... + v_k$ for some vertex v_i of P_i. $v = v_1 + ... + v_k$ is referred to as the Minkowski decomposition of vertex v.*

Proposition 2: *Let P_1,..,P_k be polytopes in $\mathbb{R}^d$ and let $P = P_1 + ... + P_k$. Let u and v be adjacent vertices of P with the Minkowski decomposition $u =$*

$u_1 + ... + u_k$ and $v = v_1 + ... + v_k$. Then u_i and v_i are either equal or adjacent in P_i for each i, and all adjacent pairs are linked by parallel edges.

The essence of these propositions is that a) the vertices of the sum polytope may be decomposed into vertices of the summands and b) the edges of the sum polytope are decomposed into either vertices or parallel edges of the summands. Figure 4.2 visualizes both propositions: The vertex $v \in R_1 \oplus R_2$ can be decomposed into vertices $v_1 \in R_1$ and $v_2 \in R_2$. Respectively, the two parallel edges e_1 and e_2 contribute to the edge e in the sum polytope, while the other two edges at v_1 and v_2 do not.

The algorithm design is based on these two observations and the fundamental idea is the following: Starting from a pre-computed vertex v^* of P, an *adjacency oracle* is used to deduce any incident edges and therefore also any neighboring point. Following a *depth-first-search (DFS)* pattern that is induced by a *local search*, the neighbors are examined iteratively until all vertices have been explored. Listing 2 abstractly describes the algorithm.

```
Input  : The summands $P_1$,...,$P_k$ in $\mathcal{V}$-representation
Algorithm:
1      $v^*$ := computeInitVertex ($P_1$,...,$P_k$);
2      $P$.addVertex ($v^*$);
3      $current := v^*$;
       repeat
4          while (current still has unexplored edges) do
5              $next$ := AdjOracle ($current$);
6              if (localSearch (next) = current) then
7                  $P$.addVertex ($next$);
8                  $current := next$;
               end
           end
9          if ($current \neq v^*$) then
10             return to the predecessor of current;
           end
11     until $current = v^*$ and all edges at current have been explored;
Output: The sum polytope $P$ in $\mathcal{V}$-representation
```

Algorithm 2: Minkowski sum computation according to Fukuda.

First, the initial vertex v^* is computed and added to the sum polytope P (lines 1 and 2). Then, a depth-first-search is started at vertex v^*, i.e., the algorithm terminates only if we return to v^* and have explored all of its edges (line 11). In context of the DFS, at every vertex v the adjacency oracle is queried (line 5) for the next neighbor. As soon as a neighbor n is found, we perform the local search on n to determine whether we proceed with our depth-first-search on n or instead consider the next neighbor of v (line 6). If the local search on n returns v, then n is added to the sum polytope P and we continue the DFS at n (lines 7 and 8). If at any time we cannot advance at a vertex v because the local search on any of its neighbors does not relate the neighbor to v again, we return to the predecessor of v with respect to the DFS (line 10).

Graphically, a directed spanning tree as depicted in Figure 4.3 with sink node v^* is constructed, where every node is a vertex of P and every edge refers to an existing edge between two vertices in P. For every node but the sink, this edge is pointed at the so-called *parent* of the node, which is determined with the local search.

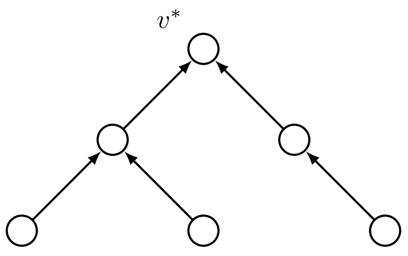


Figure 4.3: Exemplary spanning tree with v^* as a sink.

Since the algorithm requires an initial vertex $v^* \in P$ to start, which is also the sink of the spanning tree, we examine how one such vertex can be computed in the following.

For an arbitrary vector p, a unique vertex v_i may be computed for each summand P_i such that v_i is the maximum vertex within P_i in direction p. For instance, a valid approach would be to maximize v_i in each coordinate, with the priority being induced by the order of appearance. Then $v^* = v_1 + ... + v_k$ holds according to proposition 1, yielding the vertex $v \in P$ with the highest coordinates (in relation to the variable order).

Having computed v^*, the next step is to query the adjacency oracle for a neighbor, and then check if the DFS does indeed continue on this neighbor by engaging the local search.

4.1.2 Adjacency Oracle

Recalling the purpose of the adjacency oracle, it is to identify incident edges for a given vertex $v \in P$. This is achieved on basis of Proposition 2, which states that every edge of the sum polytope may be derived from the edges of the summand polytopes. However, this implies that the edges of the summands are known, i.e., for each summand P_i there has to be an adjacency list which stores the neighbors of any vertex $v_i \in P_i$. Computing this adjacency list for each P_i has a significant impact on the complexity of the Minkowski sum algorithm, as generating such a list in general is not trivial [Bur12]. Nevertheless, in Chapter 5 it will be explained why this impact is manageable in practice given the previously presented reachability analysis procedure.

Assuming the adjacency lists are available for each P_i, the adjacency oracle works as follows: First, for the given input vertex $v \in P$ its decomposition vertices v_1,...,v_k are retrieved. Then each possible edge direction may be checked individually. Without loss of generality, let p_1 be the vector that describes an incident edge of $v_1 \in P_1$ (derived from the adjacency list). First, for all vertices v_2,...,v_k it is checked whether there is a parallel edge that points in the same direction as p_1. Let all these directions be grouped together in $\Delta(p_1)$. Then the following linear feasibility problem is solved [Fuk04]:

$$\begin{aligned}
&\exists\ \lambda\ s.t.\\
&p_1^T\lambda < 0\\
&p_j^T\lambda \geq 0 \text{ for all edge directions } p_j \notin \Delta(p_1)
\end{aligned}$$

If the problem statement is feasible, i.e. a solution λ exists, then there is a hyperplane separating p_1 from the other, non-parallel edge directions and consequently p_1 determines an edge of the sum polytope P [Fuk04]. Intuitively, by requiring the scalar product $p_1^T\lambda$ to be less than zero, the vector λ points in a direction that is roughly opposite to that of p_1. In contrast, all other edge directions p_j must approximately describe the same direction as λ since $p_j^T\lambda \geq 0$ has to be satisfied. At most the p_j's may be orthogonal to λ, which is indicated by the scalar product being exactly zero.

In case of feasibility, the target vertex $v' \in P$ with the decomposition $v' = v_1' + ... + v_k'$ may be computed based on this information. If v_i contributed to the set $\Delta(p_1)$ with a parallel edge in P_i, then v_i' is set to the target of that edge, else $v_i' = v_i$. Figure 4.4 shows a slightly adjusted version of the previous Figure 4.2. Assume we have successfully solved the above LP for this instance, telling us that edge e is indeed an edge in the sum polytope $R_1 \oplus R_2$. Then we can deduce the vertex composition of v' by considering the target vertices of the parallel edges $e_1, e_2 \in \Delta(e_1)$, i.e. $v' = v_3 + v_4$. If hypothetically e_2 was not parallel to e_1, so $e_2 \notin \Delta(e_1)$, then v_2 would contribute to the decomposition of v' instead of v_4.

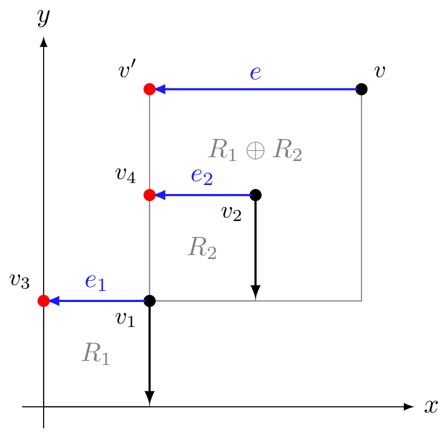


Figure 4.4: Computing the vertex decomposition of $v' \in R_1 \oplus R_2$.

By considering each edge direction induced by the summands P_i iteratively, the adjacency oracle may be used to compute all neighbors of a given vertex v in P. To decide on which neighbor n the depth-first-search is continued, it is necessary to perform a local search on each one of them as they are discovered.

4.1.3 Local Search

The local search determines the directed spanning tree that is constructed in context of the reverse search by assigning to each vertex $v \in P$ another vertex

v'' as its parent (with the exception of the sink v^*). Before examining the local search in detail, a further proposition is needed that refers to the *normal cone* $N(v,P)$ of a vertex $v \in P$ [Fuk04]:

Proposition 3: *Let v and v' be two distinct vertices of the sum polytope P and let $m_v \in N(v,P)$, $m_{v'} \in N(v',P)$ each be a vector in the normal cone of the respective vertex. Then there exists a vertex v'' adjacent to v such that $N(v'',P)$ contains a point of form $(1-\theta)m_v + \theta m_{v'}$ for some $0 \le \theta \le 1$.*

It is important to note that $v'' = v'$ may hold in this proposition. A normal cone of a vertex is indirectly defined by its incident edges and generally consists of multiple hyperplanes. The computation of the normal cone $N(v,P)$ is dependent on the dimension d of P: Let E be the ordered set of edges incident to a vertex $v \in P$. Then every sequence of $d-1$ *successive* edges in E contributes to one of $d-1$ support vectors that define one hyperplane of the normal cone. Specifically, a vector v_d describes one direction of the hyperplane if it is orthogonal to all $d-1$ edges of one edge sequence, i.e. the scalar product is 0. As an example, consider the normal cone depicted in Figure 4.5 where $d = 3$. The vertex v has four incident edges and since $d-1 = 2$, successive pairs of edges have to be considered, of which there are four considering a clock-wise order. Consequently, four support vectors may be constructed, where again every pair of those defines one hyperplane of the cone. For instance, the edges e_1 and e_2 induce one of two hyperplane support vectors v_d, with v_d being orthogonal to both source edges.

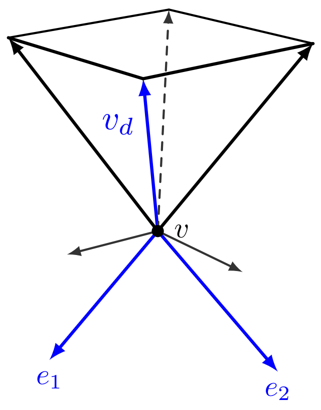


Figure 4.5: The normal cone of a vertex.

The vector m_v described in Proposition 3 may be computed by solving the following LP [Fuk04], thus also yielding an interior point of the normal cone $N(v,P)$. Let Δ be the set of all edge directions of vertices $v_1,\ldots,v_k$, $v_i \in P_i$, where $v \in P$ is decomposed into these vertices and let K be any positive constant.

$$\begin{aligned}
&\text{maximize} \quad \lambda_0 \ \textit{s.t.} \\
&p_j^T \lambda + \lambda_0 \le 0 \text{ for all edge directions } p_j \in \Delta \\
&\qquad\quad \lambda_0 \le K
\end{aligned}$$

It is not important for Δ to exclude any "false" edge directions, i.e. those that are not present in the sum polytope, because they merely provide redundant inequalities for the normal cone. The solution λ that is returned by the LP is the desired m_v, while λ_0 is a scalar and is used to assure the solution is deterministic. In particular, since the LP maximizes λ_0 the scalar product $p_j^T \lambda$ is minimized, intuitively trying to find the vector m_v that points in the most different direction with respect to each p_j.

By computing a vector $m_{v'} \in N(v',P)$ for a distinct vertex $v' \in P$ accordingly, the procedure to find the parent vertex of v can be initiated. In context of the algorithm's local search v' always refers to the sink node v^*, meaning the parent relation is defined based on the sink of the spanning tree. To identify the parent of vertex v, a ray r is shot from the target of vector m_v to the target of vector $m_{v'}$. Figure 4.6 illustrates this procedure for cones consisting of 3 hyperplanes. Both vectors m_v and $m_{v'}$ are slightly perturbed for purposes of clarity. However, perturbation may also be useful in context of degenerated polytopes such that the local search is able to identify one unique parent.

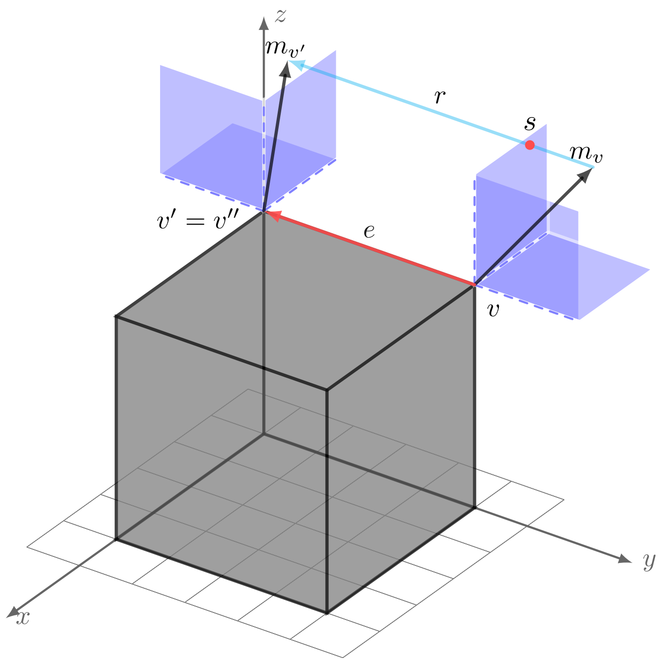
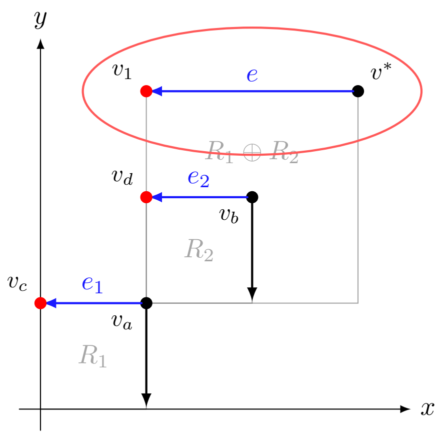


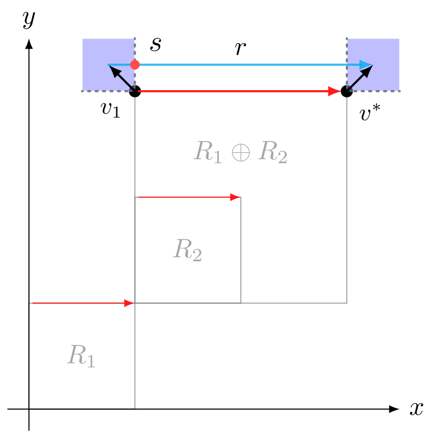
Figure 4.6: Ray shooting from m_v to $m_{v'}$.

The ray r intersects one hyperplane h of the normal cone $N(v,P)$, with intersection point s. By backtracing the computation of h, it is possible to identify one unique edge e that is characteristic for the description of the hyperplane. This edge is parallel to the normal of the intersected hyperplane and as such, instead of backtracing one can iterate through Δ and test for parallelism. Any edge $e_i \in \Delta$ that fulfills this test accurately describes the direction of an edge e incident to $v \in P$ that has $v'' \in P$ as its target. However, as decomposition edges only yield the direction but not the exact distance, the adjacency oracle is queried with direction e_i to determine the parent v'' explicitly. This step is necessary as when traversing the spanning tree according to depth-first-search, it might occur that v'' is a vertex that has not yet been observed.

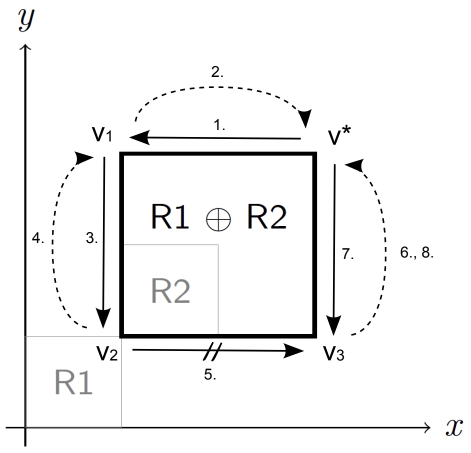
In particular, we always shoot our ray at the sink vertex $v' = v^*$, however contrary to the case that is depicted in Figure 4.6, $v' = v''$ may not always hold, i.e. for any vertex that does not have a direct edge to v^*.



(a) First adjacency oracle call.



(b) First local search call.



(c) Order in which the vertices are traversed.

Figure 4.7: Computation of the Minkowski sum according to Fukuda.

In the following, we consider the example given in Figure 4.7 to observe how the algorithm operates as a whole. Here, two polytopes R_1 and R_2 are aggregated to the sum polytope $R_1 \oplus R_2$. Vertex v^* is the pre-computed sink that is used as a starting point for the algorithm.

First, in Figure 4.7a the adjacency oracle is queried based on the edge directions at the decomposition of v^*, yielding v_1 (marked by Step 1 in Fig. 4.7c.). In Step 2 the local search is engaged to retrieve the parent of v_1 as shown in detail in Figure 4.7b. Here, two edges of the summands are parallel to the normal of the intersected hyperplane and as such v^* is the parent of v_1. Therefore v_1 is considered to be a node in the spanning tree and the DFS proceeds from here. Consequently, the adjacency oracle is asked for v_1 and returns the vertex v_2 based on the edge directions at the decomposition vertices of v_1 (Step 3).

Again, the local search on v_2 is successful and identifies the parent v_1 (Step 4), i.e. the search continues from v_2 onwards. Then, the adjacency oracle returns v_3 when queried from v_2, however the local search does not identify v_2 as its parent (Steps 5 & 6).

As v_2 has no other edges to examine, the DFS returns to the previous level in the spanning tree, which is given by vertex v_1. Again, all edges of v_1 have already been considered, such that the algorithm returns to the sink node v^*. Here, there is one further edge to explore, which is done in Step 7. The call to the adjacency oracle once again yields v_3, but this time the local search on v_3 identifies the node that the algorithm came from as its parent (Step 8). Since neither v_3 nor v^* have any more edges that are unexplored, the algorithm terminates. The spanning tree in its form after termination may be observed in Figure 4.8.

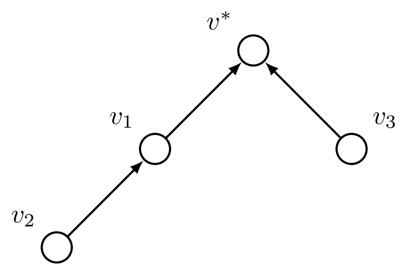


Figure 4.8: Spanning tree for the given example after termination.

Regarding computational complexity, the Minkowski sum algorithm proposed by Fukuda is dependent on solving linear programs and runs in time $O(\delta \cdot LP(d,\delta) \cdot v(P))$, whereas the required space is linear in the input size [Fuk04]. Here, δ is the aggregation of all $\delta_i, i \in 1,...,k$, where each δ_i is the maximum vertex degree occuring in P_i. $v(P)$ refers to the amount of vertices in the sum polytope P whereas $LP(d, \delta)$ denotes the required time to solve a linear program with d variables and δ inequalities, with d again referring to the dimension of the summands.

Having seen the theory behind both the reachability analysis and the Minkowski sum optimization, the next chapter introduces some details specific to the implementation of both algorithms, followed by exemplary evaluation results.

Chapter 5

Implementation & Evaluation

The purpose of this chapter is to provide insight into the implementation of both algorithms that have been presented in Chapter 3 and 4 respectively. We will also present evaluation results and briefly introduce the data model that represents hybrid automata.

The implementation took place in *HyPro* [HyP], which is a C++ library for various state set representations and computations that are involved when performing reachability analysis using these representations. *HyPro* is an ongoing, collaborative project at the university RWTH Aachen funded by the DFG.

5.1 Hybrid Automaton Data Model

The hybrid automaton data model that is used for the reachability analysis is described by the class diagram in Figure 5.1.

We can identify the following properties according to the previously given formal definition (ref. Section 2.1):

- a `HybridAutomaton` consists of:
  - a set L of `Locations`.
  - a set T of `Transitions`.
  - a set $I \subseteq L$ of *initial locations*, where currently we assume $|I| = 1$.
  - an *initial valuation* V, which is stored as a polytope.
- a `Location` has:
  - an *invariant*, which consists of a matrix, a vector and an operator.
  - an *activity matrix* A, describing the linear ODE $\dot{x}(t) = Ax(t)$ as introduced in Section 2.1.1.
  - a set of pointers to *outgoing* transitions.
  - and lastly a matrix E for *external input*. The purpose of this matrix is to support non-autonomous systems (see Section 2.1.2), which are not in the scope of this thesis, but may be part of future work.
- a `Transition` has:

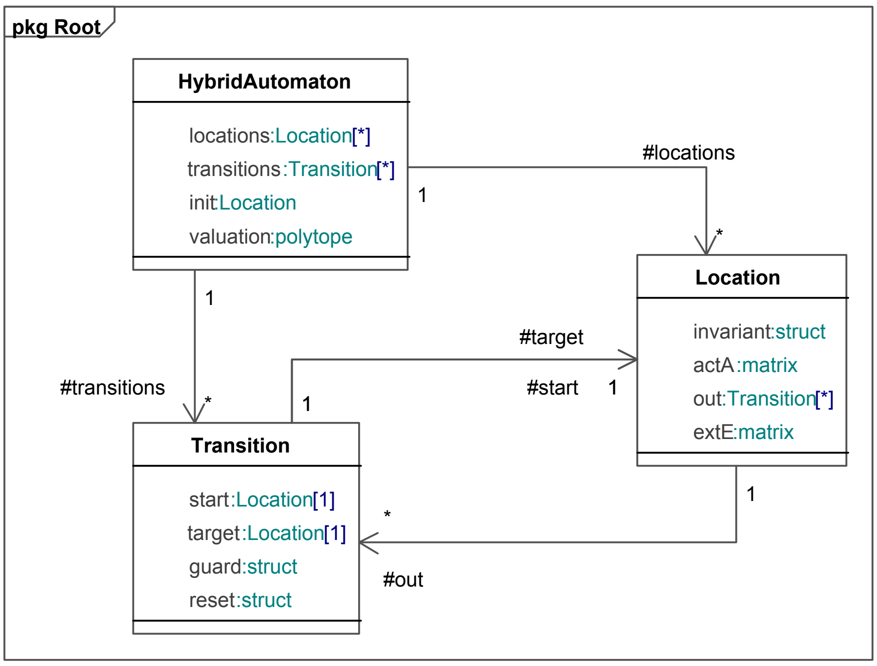


Figure 5.1: Class diagram of the hybrid automaton data model. A hybrid automaton consists of multiple transitions and locations, as well as one initial location and a variable valuation in form of a polytope.

- a pointer to its *start and target* locations.
- a *guard*, which again consists of a matrix, vector and operator.
- a *reset*, implemented in form of a transformation matrix and an optional translation vector.

In order to store matrices and vectors as well as perform operations on them, *HyPro* uses the library *Eigen3* [Eig]. *Eigen3* is a C++ template library for linear algebra and provides the `MatrixBase` class from which matrices and the like may be derived.

In the given data model, the activity matrix A is of particular importance as it determines the behaviour of the flowpipe that we want to compute. As previously alluded in Section 2.1.1, the activities of a linear hybrid automaton may also contain a constant part b: $\dot{x}(t) = Ax(t) + b$. Since the recurrence relation we used in Chapter 3 for approximating the flowpipe segments is based on equations of the form $\dot{x}(t) = Ax(t)$, we have to encode b into a new matrix A'. Equation 5.1 illustrates how this is done for an exemplary ODE.

$$\dot{x}(t) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} x(t) + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad \Longrightarrow \quad \dot{x}(t) = \begin{pmatrix} 1 & 0 & b_1 \\ 0 & 1 & b_2 \\ 0 & 0 & 0 \end{pmatrix} x(t) \qquad (5.1)$$

A' has one additional row and column when compared to A, which are used to encode the constant factor b. In particular, a new artificial dimension and thus also a variable is introduced into the model. Let x_3 be this variable. We require the row in A' that corresponds to x_3 to be a strict zero row, and by setting $x_3(0) = 1$ initially we get the system of linear equations as illustrated in Equation 5.2.

$$\begin{aligned} \dot{x_1}(t) &= x_1 + x_3 * b_1 = x_1 + b_1 \\ \dot{x_2}(t) &= x_2 + x_3 * b_2 = x_2 + b_2 \\ \dot{x_3}(t) &= 0 \end{aligned} \tag{5.2}$$

We can see that by setting x_3 to 1 and keeping it at that value independently of how much time passes, we can emulate a constant factor b that is added onto the variables x_1 and x_2.

However, it is important to note that x_3 is only considered to be an artificial dimension. Therefore we have to explicitly exclude it for certain parts of the reachability algorithm, i.e. when bloating a first approximation based on the computed upper bound for the Hausdorff distance. Since we do not want to bloat in dimension x_3, we have to be aware of whether such an encoding took place or not.

After this brief introduction to how hybrid automata are stored and processed in *HyPro*, the next section will deal with some implementation details of the reachability algorithm.

5.2 Reachability Algorithm

Let us recall the general reachability algorithm that has been presented in Chapter 3:

```
Input  : Set $Init$ of initial states
Algorithm:
1     $R_{new} := Init$;
2     $R := \emptyset$;
3     while ($R_{new} \neq \emptyset$) do
4         $R := R \cup R_{new}$;
5         $R_{new} :=$ Reach ($R_{new}$)$\backslash R$;
      end
Output: Set $R$ of reachable states
```

Here, `Reach(..)` is called on a set of states and computes those states that are reachable from the given set. Since this reachability relation includes both continuous as well as discrete steps, in the implementation the functionality is split into two sub-functions: `computeTimeClosure(..)` and `computePostCondition(..)`.

The function `computeTimeClosure(..)` computes the flowpipe approximation for one specific location. Its implementation is abstractly described by Listing 3.

```
**Input** : One location $l$, a corresponding valuation $X_0$ and a timebound $t_{bound}$.
**computeTimeClosure()**
1       $t_s := t_{bound}/discretizationFactor$;
2       $InitApprox :=$ convexHull $(X_0 \cup e^{t_s A} X_0)$;
3       $\alpha_{t_s} :=$ computeHausdorffBound($X_0, t_s, A$);
4       $Bloating :=$ Ball($\alpha_{t_s}$);
5       $\Omega_0 := InitApprox \oplus Bloating$;
6       $lastSegment := \Omega_0$;
7       **for** $(i \leq t_{bound})$ **do**
8           **if** (lastSegment not in invariant of $l$) **then** abort ;
9           $F$.addSegment($lastSegment$);
10          $lastSegment = e^{t_s A}(lastSegment)$;
11          $i += t_s$;
        **end**
**Output**: Flowpipe approximation $F$ for location $l$.
```

Algorithm 3: Computing the flowpipe approximation of one location.

The function first approximates the initial flowpipe segment Ω_0 according to the procedure presented in Chapter 3: $\Omega_0 = CH(R_0(X_0) \cup R_\delta(X_0)) \oplus B(\alpha_\delta)$.

To do so, we first compute our approximation step size $t_s = \delta$ in line 1 of Listing 3. Since the valuation X_0 is stored as a polytope, the initial approximation can be computed (line 2), which is bloated based on an upper bound for the Hausdorff distance (lines 3 and 4). The result of the Minkowski sum then yields the final approximation of Ω_0 (line 5). In lines 7 to 11 we compute the following flowpipe segments iteratively, until either the timebound t_{bound} is reached or the invariant of location l is violated, causing the computation to stop. It is noteworthy to mention that the computation of the matrix exponential is currently delegated to the *Eigen3* library, which is beneficial in the sense of reliability, but does not leave any space for optimization or parametrization.

The function `computePostCondition(..)` on the other hand computes the reachability in context of one discrete step, i.e. one transition in the automaton. It is called on each flowpipe segment of one location individually, first asserting whether the transition's guard is fulfilled. Again, Listing 4 conveys the basic idea.

In line 1 the intersection between the given flowpipe segment and the guard of the considered transition is computed. This boils down to intersecting either two polytopes, or one polytope (the flowpipe segment) and a hyperplane (the guard). If this intersection is empty, intuitively it is not possible to make this discrete step given the variable valuation as defined by the flowpipe segment (line 5). However, if the intersection is not empty the reset of the transition is applied to this intersection, which is a linear transformation followed by an optional translation (line 3). The result is a new valuation *Val* in the target location of t, which is stored as a polytope.

Since `computePostCondition(..)` only processes individual flowpipe segments, the function that calls it has to provide the scope of a whole, connected flowpipe. As outlined in Chapter 3, if multiple flowpipe segments have a non-empty intersection, all the returned valuations are united in the target location.

```
Input  : One flowpipe segment $P$ and one transition $t$.
computePostCondition()
1      $Intersection := P \cap t.\texttt{guard}()$;
2      if ($Intersection \neq \emptyset$) then
3          $Val := t.\texttt{applyReset}(Intersection)$;
4          return $Val$;
       end
       else
5          return $false$;
       end
Output: False or a new $X_0$ in the target of $t$.
```

Algorithm 4: Application of one discrete step to a flowpipe.

By applying the convex hull thereafter, we obtain the new initial state set X_0, which may be used to approximate one flowpipe of the new location.

In the following, results of the algorithm shall be presented.

5.2.1 Reachability Evaluation

In this section we will focus on the reachability in context of one location (disregarding any discrete steps), as this is where the major work is done and where various approaches in literature differ.

First, a brief toy example shall illustrate how the reachability approximation result compares to the exact flowpipe. Let $X_0 = \{(0,0)^T, (0,1)^T, (1,0)^T, (1,1)^T\}$ be the initial valuation representing a two-dimensional box of width 1. As any box is also a polytope, X_0 is a valid input for the reachability algorithm. Furthermore, let us assume we want to scale this box by a factor of two in each time-unit. Then the linear ODE is given as follows:

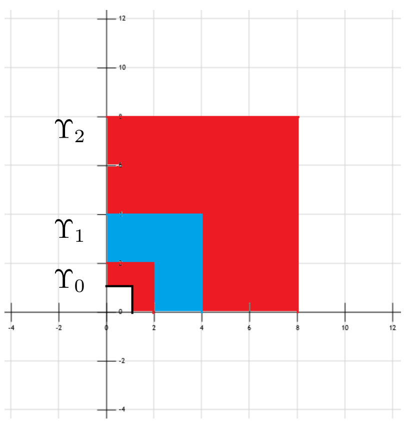
$$\dot{x}(t) = \begin{pmatrix} ln(2) & 0 \\ 0 & ln(2) \end{pmatrix} x(t) \tag{5.3}$$

Here, we do not include any constant part b, such that our matrix A describes only the two dimensions that are indeed part of the example. For the step size we choose $\delta = 1$ for purposes of clarity. Also, we set $x_1 < 16 \wedge x_2 < 16$ as the invariant, which limits the flowpipe size to three segments given $\delta = 1$, since we don't count X_0 to be a segment of its own.

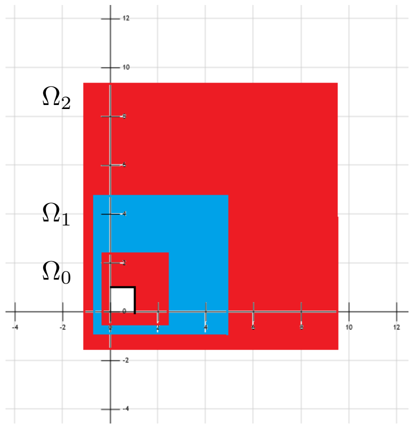
Then the upper bound for the Hausdorff distance yields the result:

$$\begin{aligned} \alpha_\delta &= (e^{\delta||A||} - 1 - \delta||A||) \max_{x \in X_0} ||x|| \\ &\approx 0.306852 \end{aligned}$$

The exact flowpipe for this exemplary setup is given in Figure 5.2a, where the initial box X_0 is filled white. We can see that the flowpipe consists of three segments, each being indicated by a different colour, starting with red. However, it has to be pointed out that each segment also completely contains the previous one due to the setup, which may not be visible in the figure. As a comparison, the result of the reachability algorithm is illustrated in Figure 5.2b.



(a) Exact flowpipe.



(b) Approximated flowpipe.

Figure 5.2: Comparison between approximation and exact result.

First, we may observe that the approximation also consists of three flowpipe segments Ω_0, Ω_1 and Ω_2. The approximation of the first flowpipe segment Ω_0 is essential, as the error here is accumulated in the later segments. When comparing both figures with each other, it is obvious that Ω_0 is an over-approximation of the exact set Υ_0 in Figure 5.2a. In particular, one vertex of Υ_0 is (2,2), whereas the equivalent vertex for Ω_0 is $(2+\alpha_\delta, 2+\alpha_\delta) \approx (2.3069, 2.3069)$. Furthermore, since the bloating has been performed on the basis of a 2-dimensional hypercube, there is also an over-approximation into the negative directions. For instance, the vertex $(-\alpha_\delta, -\alpha_\delta) \approx (-0.3069, -0.3069)$ is also part of Ω_0.

Nevertheless, as the exact flowpipe is generally not computable for hybrid systems [HKPV95], an over-approximation on basis of the presented reachability algorithm still allows us to derive statements about the reachable state set. For instance, if a safety property holds on the over-approximation then it also holds on the exact solution.

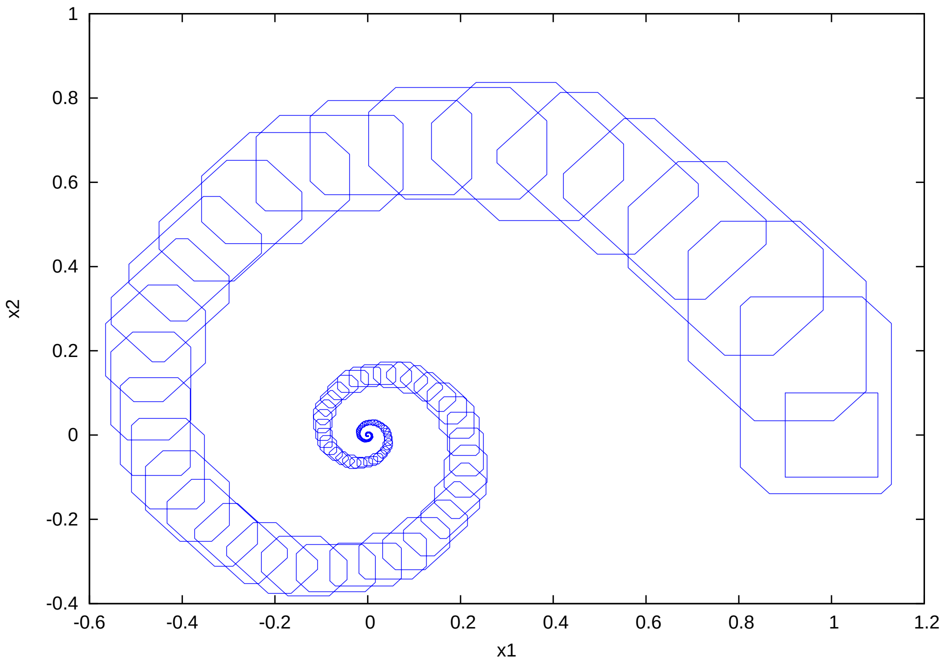
In the following the result of the reachability algorithm that has been implemented in context of this thesis shall be compared to the computations of the tool *Flow**, which will be introduced in detail in Chapter 6.

The initial valuation of the test case is again described by a box $X_0 = \{(0.9, -0.1)^T, (0.9, 0.1)^T, (1.1, -0.1)^T, (1.1, 0.1)^T\}$, whereas the ODE is given by:

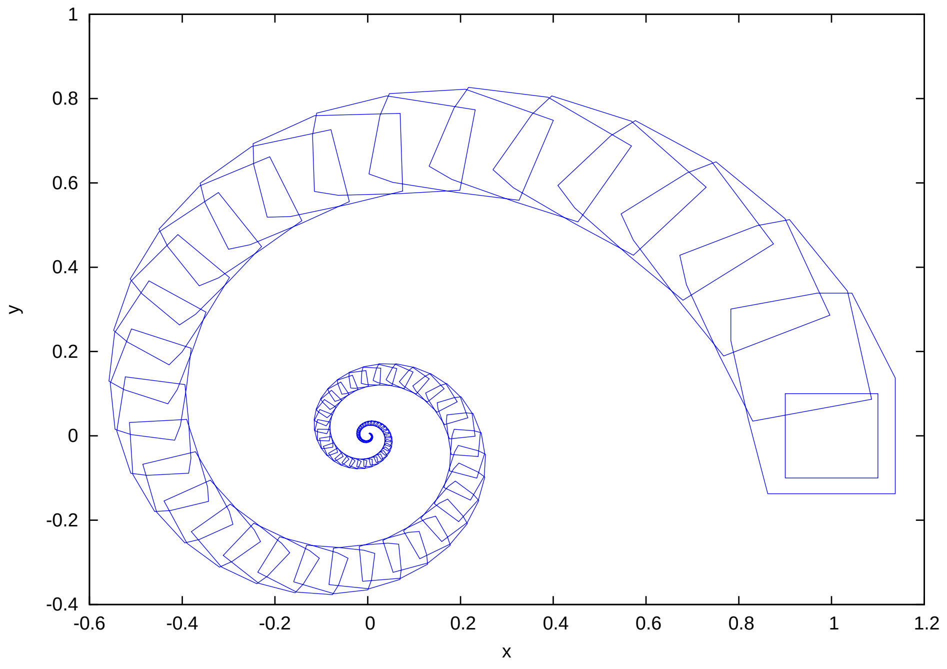
$$\dot{x}(t) = \begin{pmatrix} -1 & -4 \\ 4 & -1 \end{pmatrix} x(t) \tag{5.4}$$

The time horizon is set to 5 seconds and the employed stepsize is 0.05, resulting in a total of 100 flowpipe segments for the approximation. The output of both algorithms is shown in Figure 5.3, where *GnuPlot* [Gnu] has been used for visualization.

The initial valuation box is included within the first approximated flowpipe segment and as a whole, both approximation results are reasonably similar. The major difference lies in the set representation that is used in each case: Whereas *Flow** is based on octagons, our implementation yields polytopes with

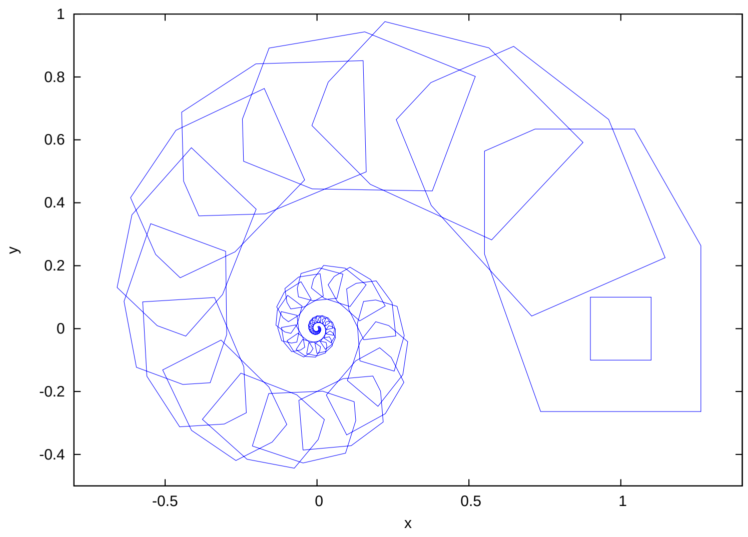


(a) Output of Flow*, using octagons for approximation.

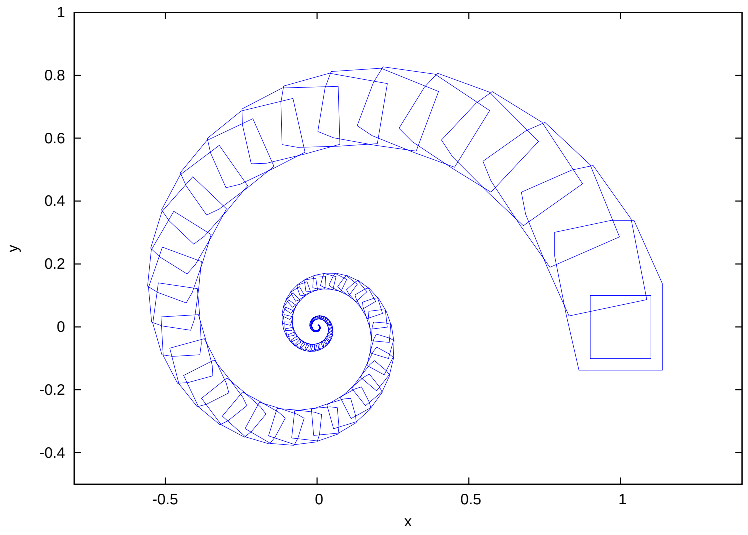


(b) Output of our implementation, making use of polytopes.

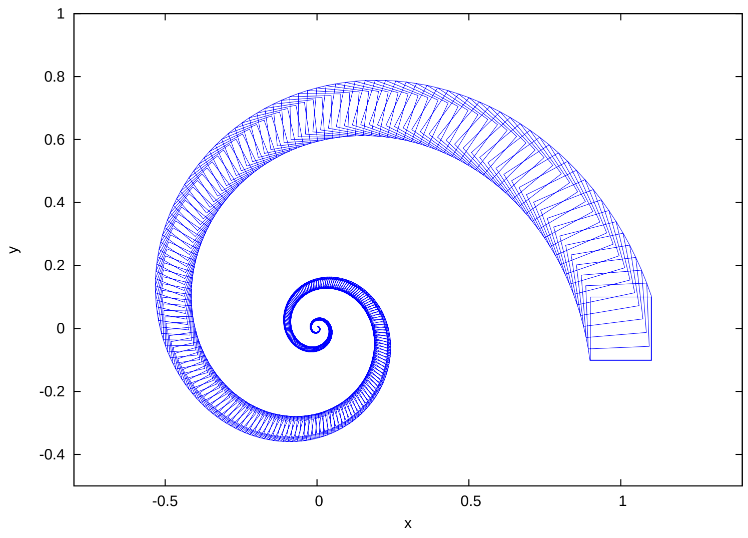
Figure 5.3: Comparison between two reachability approximation approaches. Both plots have been generated with *GnuPlot*.



(a) Step size: 0.1 | Segments: 50 | $\alpha_\delta = 0.1636$ | Computation time: 0.023s



(b) Step size: 0.05 | Segments: 100 | $\alpha_\delta = 0.0374$ | Computation time: 0.073s



(c) Step size: 0.01 | Segments: 500 | $\alpha_\delta = 0.0014$ | Computation time: 0.155s

Figure 5.4: Impact of differering step sizes. The time horizon is always 5, whereas individual parameters for each case are listed below the figures. Here α_δ refers to the computed Hausdorff distance estimate.

eight vertices each as well, although they are rather hard to identify. In case of computation time, both approaches required less than 0.1 seconds including writing the data to an output file.

In Figure 5.4 the impact of different step sizes is illustrated: Subfigure 5.4a shows the result for a step size of 0.1, while Subfigures 5.4b and 5.4c each provide a refinement. A decrease in the step size correlates to an increase in the total amount of flowpipe segments and as such the approximation is more accurate, albeit at the cost of taking longer.

Having seen how the reachability algorithm performs, we continue with some implementation details of the Minkowski sum algorithm.

5.3 Minkowski Sum

Before evaluating the implemented Minkowski sum algorithm, in the following some specific deviations from its original form as suggested in [Fuk04] will be introduced.

Summation of k polytopes

The approach proposed by Fukuda is tailored for the simultaneous addition of k polytopes P_1,...,P_k. In contrast, the implementation in this thesis is restricted to the addition of two polytopes at one time. While an addition of k polytopes may be simulated by sequentially adding each polytope onto the sum of the previous iteration, there might be some computational concerns when doing so as outlined in [Wei07].

In particular, summing all polytopes in a single computation has been extensively compared against an incremental scheme. The restraining factor of the latter has been identified to be the adjacency list that is required at initialization of the algorithm: Instead of computing the adjacency list for each of the k summands once, the iterative approach required a recomputation of the adjacency list for each intermediate sum, which took significantly more time than solving the Minkowski sum computation itself [Wei07].

Therefore, in this thesis Fukuda's algorithm has been extended to also compute the adjacency list of the sum polytope during execution. This is done as soon as a neighboring vertex is discovered by the adjacency oracle. While this requires a greater amount of memory to store intermediate sum polytopes, the computational complexity stays the same.

As suggested in [Wei07], considering this adaptation an incremental computation of the sum of k polytopes may even be beneficial. Furthermore our application area, which is the reachability algorithm, only requires two polytopes to be aggregated at the same time, which happens in context of bloating the first, possibly under-approximating polytope.

Indeed, following an incremental scheme in the context of the reachability algorithm we only need to compute exactly two adjacency lists per flowpipe: The first one for the initial polytope X_0 that is used as a basis for the flowpipe approximation and the second one for the hypercube which is used for bloating. As the Minkowski sum algorithm is extended to update the adjacency list of the sum polytope as soon as vertices are discovered, our initial flowpipe segment Ω_0 already stores all neighbor information that is needed. Since later segments are only computed by application of a linear transformation to the previous segment, we can preserve the adjacency list by extending the linear transformation procedure.

Therefore, considering the given context, having to compute the adjacency list for a polytope before application of the Minkowski sum has a rather small impact on the efficiency of the reachability algorithm.

Optimization of local search frequency

A further optimization to the Minkowski sum algorithm that took place in this thesis is in regard to how often the local search is engaged. The original proposal by Fukuda in [Fuk04] suggests a redundant use of the local search function when determining to which node the algorithm returns in the spanning tree as soon as one branch has been completely examined. Instead of engaging the local search to query for the parent of the already processed node, we store parents of nodes in a map as soon as they have been computed once. Therefore, the local search is not unnecessarily engaged multiple times on the same vertex, which may result in up to $n - 1$ less calls to the function (where n is the amount of nodes in the spanning tree).

5.3.1 Performance Comparison

In the following, the implemented Minkowski algorithm (also referred to as *reverse search*) shall be compared to a brute force implementation. Here, brute force again refers to the method of aggregating every vertex of one summand with every vertex of the other summand, followed by an application of the convex hull (ref. Section 4.1). The goal is to examine whether the proposed implementation is beneficial for complex hybrid systems that often occur in practice, where 10+ variables are involved and consequently polytopes of relatively high dimension with a multitude of vertices have to be aggregated.

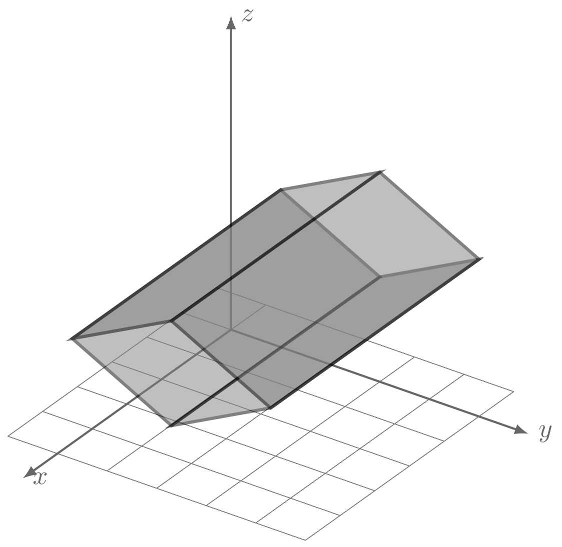


Figure 5.5: Setup for the vertex test of the Minkowski sum.

For the evaluation, we assume the adjacency list of the summand polytopes to be pre-computed, as we previously identified this requirement to have a negligible impact given the reachability analysis context. All computations have been performed on a commercially available Intel I7 notebook. Specifically, up to 4 GB of RAM and four cores each with a CPU frequency of 1.6 GHz were available.

Dimension	#Vertices	Reverse Search	Brute Force
3	8	0.031	0.002
3	20	0.072	0.031
3	100	0.404	2.705
3	200	0.985	24.119
3	500	3.756	708.690
3	1000	11.809	-
3	5000	229.142	-

Table 5.1: Vertex test results: computation time in seconds. For upwards of 100 vertices the performance of the reverse search was significantly better.

The first evaluation example consists of the addition of two polytopes that each have the form of a prism and will be referred to as the *vertex test*. One summand is conceptually depicted in Figure 5.5, whereas the second summand is just a shifted copy of the first one. In this test case the dimension d is always three, whereas the amount of vertices that the prism consists of is variable. It is noteworthy that the prism will always retain its symmetry, i.e. vertices will be lost or gained at both of its sides.

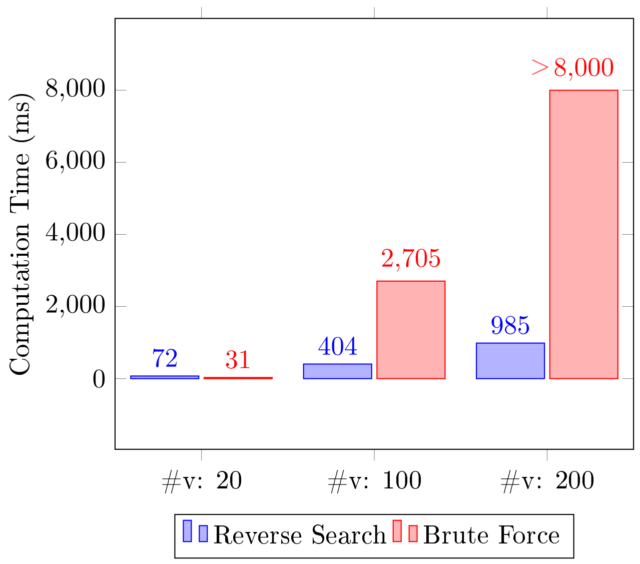


Figure 5.6: Bar chart that visualizes the results for the vertex test.

There are multiple reasons for why this specific example setup has been chosen. First, the prism structure allows for pre-computation of the adjacency list with relatively low effort and secondly, a moderate amount of inner points are discarded in context of the brute force algorithm. The results are an average over three iterations of the same computation and can be observed in Table 5.1.

While the brute force approach performs better for a low amount of vertices, the reverse search algorithm is distinctly beneficial when it comes to higher amounts. In fact, for more than 500 vertices it does not seem feasible to use the brute force computation anymore, since the needed computation time already exceeded 10 minutes, which is a strong contrast to the 3.8 seconds that the reverse search required. Consequently, the measurements for 1000 and 5000 vertices have only been performed for the reverse search algorithm. Still, even for 5000 vertices the reverse search took less time than the brute force approach required for 500, suggesting a great scalability with respect to the total amount of vertices in a polytope. Figure 5.6 summarizes the results of the comparison in context of a bar diagram.

A further test has been conducted where both the vertices and the dimension are variable (albeit we will still refer to it as the *dimension test*). The test setup consists of two unit hypercubes, where again the second summand is a shifted version of the first one. In general, for dimension d a unit hypercube has 2^d vertices and each vertex has d neighbors. Two exemplary hypercubes for dimensions 2 and 3 respectively are shown in Figure 5.7.

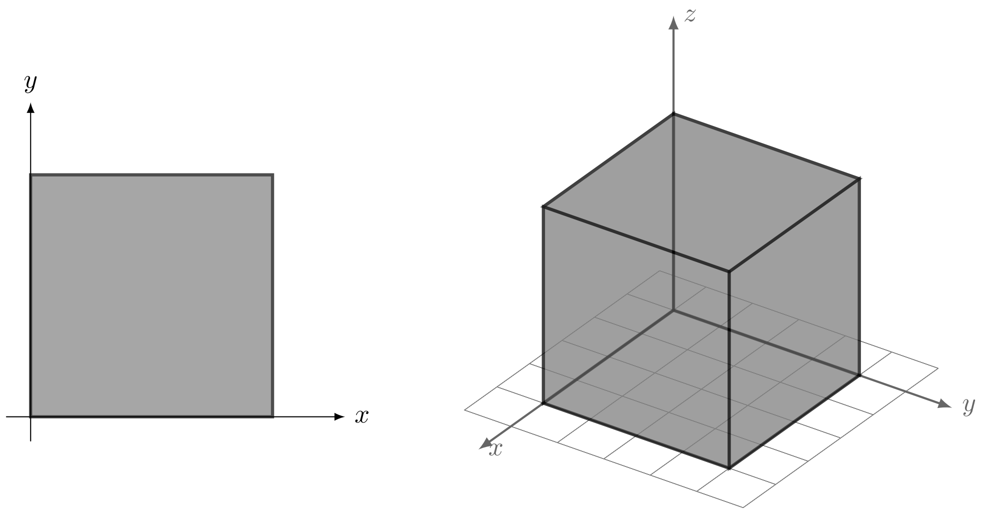


Figure 5.7: 2-dimensional hypercube (left) and 3-dimensional hypercube (right).

Again, results may be observed in Table 5.2 and reflect the average over three measurements. Initially, the brute force algorithm is more competitive than in the vertex test when compared to the reverse search. However, as both the dimension and the amount of vertices increase, the computation time for the brute force approach rises to significant heights again. The bar diagram in Figure 5.8 illustrates the results graphically.

Intuitively, it is reasonable to expect that the brute force approach scales comparatively better for higher dimensions, while the reverse search algorithm scales better for a high amount of vertices. The reason being that the brute force method aggregates every vertex in one set with every vertex in the other set, i.e. there is a high computational effort for a lot of vertices, but only changing the dimension does not impact the performance as much.

Dimension	#Vertices	Reverse Search	Brute Force
2	4	0.005	0.001
4	16	0.106	0.018
6	64	1.329	1.203
8	256	12.015	100.266
9	512	43.327	874.063
10	1024	123.983	-
11	2048	364.687	-

Table 5.2: Dimension test results: computation time in seconds. From dimension 6 onwards the reverse search performed either comparably or significantly better.

In contrast, the reverse search enumerates the vertices of the sum, such that an increasing amount of vertices has a lesser effect on the required computation time. In case of an increasing dimension however, vertices often also gain additional neighbors. As outlined in Chapter 4, the linear programs that are solved in context of the reverse search depend on the amount of edges of the vertex decompositions. Since those increase if a decomposition vertex gains additional neighbors, it is reasonable to expect the reverse search algorithm to perform not quite as well for an input of high dimension. However, as a side effect we gain the neighborhood information without investing further computational effort.

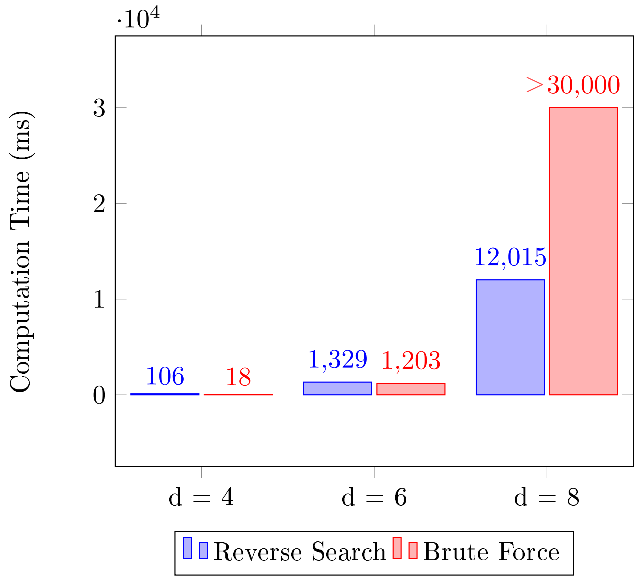


Figure 5.8: Bar chart that visualizes the results for the dimension test.

Altogether, we have seen that the implemented algorithm for the Minkowski sum is a significant improvement over a brute force approach. Since the Minkowski sum is required in context of the reachability algorithm, we can expect to harvest benefits there as well.

Chapter 6

Related Work

This chapter gives insight into some related work for the complex topic of reachability analysis in context of hybrid systems. First, an overview over existing approaches is given, followed by the introduction of some tools that implement reachability algorithms.

6.1 Reachability Analysis in Literature

In literature, there is a fair amount of approaches that also make use of polytopes to approximate reachable state sets.

For instance, Krogh et al. [CK98] propose a procedure that is very similar to the one implemented in this thesis. However, instead of relying on the Hausdorff distance as a metric to determine a reasonable over-approximation, Krogh et al. consider linear optimization problems to minimize the approximation error. In order to solve those, it is necessary to simulate executions of the underlying hybrid system, e.g. with MATLAB [CK98].

Another approach suggested by Alur et al. in [ADI02] combines the idea of predicate abstraction with the polyhedral approximation of reachable sets. The general notion is to compute an abstraction of the hybrid system based on user-supplied predicates, where the continuous variables in the origin system are replaced by discrete boolean variables. A state in the abstraction corresponds to an existing truth assignment of the input predicate, such that if the abstraction satisfies a certain property, so does the original system [ADI02]. To analyze the reachability of the system, a verifier performs a search on the abstraction by manipulating the predicates, which are stored as polytopes. However, the drawback of the presented approach is that the procedure is reliant on the quality of the input predicates.

Leaving the realm of polytopes, there are various algorithms that make use of other set representations to approximate the reachable set of hybrid systems, some of which will be briefly introduced in the following.

Zonotopes

In Section 2.2.2 zonotopes have been introduced as a sub-class of polytopes with special properties. The main benefit of using zonotopes over polytopes to approximate reachable state sets lies in the efficiency of the implementation

[ADF$^+$06]. Since most approximation methods involve the Minkowski sum, polytopes cause an increase in representation complexity at each aggregation. As zonotopes may be represented as the Minkowski sum of line-segments [LG09], the Minkowski addition of two zonotopes can be computed very efficiently. However, zonotopes have their own drawback in the context of reachability analysis: Each successive linear transformation that is applied on a zonotope causes an increase in its generators, leading to the computations becoming intractable for large time bounds [ADF$^+$06]. A solution to this problem has been proposed in context of [Gir05], where a zonotope of higher order is over-approximated by a zonotope of lower order and thus the representation complexity can be reduced.

Concluding, zonotopes are a considerable alternative to polytopes when approximating reachable state sets. While polytopes may generate more precise approximations especially for larger time horizons, zonotopes allow for a very efficient and scalable implementation.

Besides approximating reachable state sets by geometric objects, there is also the possibility of using support functions.

Support functions

Often, support functions allow for a closer approximation of the first flowpipe segment compared to the alternatives [LG09], however there are frequently accuracy problems in context of discrete transitions, where the guard of a transition (often a hyperplane) has to be intersected with the convex set that is defined by the support function. Therefore, in [Ray12] an alternative approach to compute the support function of the intersection is proposed, which reduces the task to a convex minimization problem.

Non-linear hybrid systems

Finally, all previous considerations as well as the implementation that took place in context of this thesis were targeted at linear hybrid systems. In case of non-linear systems, where the dynamics are defined by non-linear differential equations, the reachability analysis is comparatively harder. A proposed approach in this context is often referred to as *hybridization*, where the state space is split into small disjoint regions [ADF$^+$06]. For each region, a primitive piecewise approximation of the system is computed and then the reachability analysis is initiated on these region approximations.

6.2 Existing Tools

In the following, two tools that implement reachability analysis algorithms are introduced briefly.

6.2.1 SpaceEx

SpaceEx is a tool that serves the purpose of performing safety analysis for hybrid systems by computing the reachable state set [FLGD$^+$11]. To do so, it combines both support functions as well as polytope representations to compute the over-approximation. *SpaceEx* has been proven to be capable of handling fairly complex systems, with up to 200 variables [Ray12]. The hybrid system that is to be analyzed may be specified as a network of hybrid automata and the model is stored in a proprietary file format. Furthermore, *SpaceEx* also supports the analysis of some non-linear hybrid systems by incorporating the approach of hybridization as outlined previously.

6.2.2 Flow*

*Flow** [Flo] is a fairly new tool that has been released in November 2013. The tool supports hybrids systems with non-linear ODEs as well as uncertain inputs and computes *Taylor model* flowpipes. The approximation step sizes are adaptively changed during the flowpipe construction and the visualization is done in context of a projection on a two dimensional plane.

Chapter 7

Conclusion

7.1 Summary

In this thesis, methods to approximate the reachable state set of linear hybrid systems have been explored. After a brief introduction to existing set representations and the operations that need to be performed on these sets, the general reachability algorithm for autonomous systems has been introduced. The presented algorithm uses polytopes to approximate state sets and relies on the Hausdorff distance to guarantee an over-approximation.

Furthermore, optimization opportunities for the reachability analysis have been investigated. In particular, since the main hindrance for the polytope based analysis relates to the computation of the Minkowski sum, a sophisticated approach that enumerates the vertices of the sum polytope has been implemented. The results of the evaluation propose that the sum implementation is able to handle even complex hybrid systems, while the reachability algorithm provides accurate over-approximations of the reachable state sets for at least the tested examples.

Nevertheless, there is a lot of room for improvement and additional features in context of future work.

7.2 Future Work

One of the most important tasks is the extension of the reachability algorithm to support non-autonomous systems as introduced in Section 2.1.2. As the hybrid automata data model already considers an optional user input, what is left is to adjust the recurrence relation according to which the flowpipe segments are computed. In particular, a set V that accounts for the influence of the uncertain inputs has to be added onto the previously computed segments [ADF+06]:

$$\Omega_{i+1} = e^{\delta A}\Omega_i \oplus V \tag{7.1}$$

As this new recurrence relation implies the use of the Minkowski sum at every computed flopwipe segment, it is of even greater benefit that we found a scalable solution that has already been implemented successfully.

Furthermore, some other operations may be optimized to improve the computational efficiency of the reachability algorithm. Notably, the computation of the convex hull seems to provide the opportunity to harvest significant benefits as outlined in [AB95].

Also, currently visualization for the the computed flowpipe approximations is only available for two dimensional input models. An extension to higher dimensions is desireable, where a projection of the results to a 2D or 3D plane is possible in context of *GnuPlot* [Gnu].

Bibliography

[AB95] David Avis and David Bremner. How good are convex hull algorithms? In *Proceedings of the eleventh Annual Symposium on Computational Geometry*, pages 20–28. ACM, 1995.

[ADF$^+$06] Eugene Asarin, Thao Dang, Goran Frehse, Antoine Girard, Colas Le Guernic, and Oded Maler. Recent progress in continuous and hybrid reachability analysis. In *Proceedings of the IEEE International Symposium on Computer-Aided Control Systems Design*, pages 1582–1587, 2006.

[ADI02] Rajeev Alur, Thao Dang, and Franjo Ivančić. Reachability analysis of hybrid systems via predicate abstraction. In *Hybrid Systems: Computation and Control*, pages 35–48. Springer, 2002.

[Ábr12] Erika Ábrahám. Modeling and analysis of hybrid systems. Lecture, RWTH Aachen University, Summer term 2012.

[Bur12] Benjamin A Burton. Complementary vertices and adjacency testing in polytopes. In *Computing and Combinatorics*, pages 507–518. Springer, 2012.

[CK98] Alongkrit Chutinan and Bruce H Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *Proceedings of the 37th IEEE Conference on Decision and Control*, volume 2, pages 2089–2094, 1998.

[Eig] Eigen3. Linear algebra library. http://eigen.tuxfamily.org/, September 2014.

[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.

[Flo] Flow*. http://ccis.colorado.edu/research/cyberphysical/taylormodels/, September 2014.

[Fuk04] Komei Fukuda. From the zonotope construction to the Minkowski addition of convex polytopes. *Journal of Symbolic Computation*, 38(4):1261–1272, 2004.

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

[Gnu] GnuPlot. http://www.gnuplot.info/, September 2014.

[GS93] Peter Gritzmann and Bernd Sturmfels. Minkowski addition of polytopes: Computational complexity and applications to Gröbner bases. *SIAM Journal on Discrete Mathematics*, 6(2):246–269, 1993.

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

[HyP] HyPro. http://ths.rwth-aachen.de/research/hypro/, September 2014.

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

[MVL03] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM review*, 45(1):3–49, 2003.

[Ray12] Rajarshi Ray. *Reachability Analysis of Hybrid Systems Using Support Functions.* PhD thesis, Université de Grenoble, 2012.

[Str03] Gilbert Strang. Introduction to linear algebra. *Cambridge Publication*, 2003.

[Wei07] Christophe Weibel. *Minkowski sums of polytopes: combinatorics and computation.* PhD thesis, École Polytechnique Fédérale de Lausanne, 2007.

[Zie95] Günter M Ziegler. *Lectures on polytopes*, volume 152. Springer, 1995.