Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

MASTER OF SCIENCE THESIS

OPTIMIZING REACHABILITY ANALYSIS FOR NON-AUTONOMOUS HYBRID SYSTEMS USING ELLIPSOIDS

Phillip Florian

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Additional Advisor:
Stefan Schupp

Aachen, September 28, 2016

Abstract

Hybrid systems are systems with a mixed discrete and continuous behavior. Due to their increasing occurrence in industry and science hybrid systems are often safety critical. Because of that, a lot of effort was and still is put into developing various algorithms and tools for hybrid systems verification.
In this thesis we give an introduction to hybrid systems and to flowpipe-based reachability analysis of those systems. We specially focus on optimizing this reachability analysis in the presence of support functions by reducing their complexity. Further, we propose a new way of computing the set of reachable states on hybrid systems with external influence by usage of ellipsoids. We then evaluate our optimizations showing the general applicability of our approach and make some suggestions for further research in this area.

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Acknowledgments

At this point I would like to thank all the people that made my whole studies an awesome time. First of, I have to thank my parents, my girlfriend and especially Tim for for all the support they gave me. Without them I would probably have dropped out at some point. Further, I would like to thank everyone working at the *i*2 chair here at RWTH Aachen for their support, advices and knowledge which they provided over the last 5 years.

Finally, a special thanks to my advisor Stefan Schupp who guided me trough the work on this thesis and always had time to help me solve the problems that were emerging during the last semester of my studies.
You are all awesome people and I doubt that I would stand where I am today if it was not for you.

Thank you.

CONTENTS

CHAPTER 1

INTRODUCTION

Hybrid systems are systems with a combined discrete and continuous behavior. Typical examples are physical systems controlled by discrete controllers which can be found in various places like aviation, control engineering or medicine, just to name a few [ACHH93, DA01, GG09].
Due to the increasing number of hybrid systems in the fields stated above, safety verification for those systems becomes more and more important. Therefore, an increasing request for formal methods for verification can be observed.
Purely discrete systems as well as purely continuous systems have already been researched for years and thus, there are well-established ways of verification available for those systems. For hybrid systems however there are just few verification techniques with a lot of space for improvements due to the relatively young age of this field of research.

Reachability analysis of hybrid systems has been a major field of research in hybrid systems for over a decade now and it is still of interest. Currently there are three different approaches for reachability analysis. The approaches are SMT-solving, theorem proving and flowpipe computation [GT08, GG09]. Flowpipe-based approaches are often using geometric objects to represent reachable sets including polyhedrons and ellipsoids or symbolic representations, such as support functions or Taylor models [GGM06].

Independent of the techniques used for reachability analysis, the goal is always the same. Given a system we want to verify whether a set of bad states is reachable. A general approach is to start with the initial states of the system and observe the evolution of the system in an iterative way. In each such state we then check if the properties are satisfied. If we discover a bad state to be reachable we call the system unsafe. If non such state is reached we call it safe.

Reachability analysis for hybrid systems is even more complex than for most other systems due to the combination of discrete and continuous behavior. The system can either stay in its current mode for a certain time or change its mode instantly. In the first case the state of the system evolves over time, in the second case the state

changes instantly. Due to this continuity the reachability problem becomes undecidable for some classes of hybrid systems [HKPV95]. Thus, safety verification of hybrid systems is a complex field with many aspects to take into account.

At the current state, the most one of the most popular tool is SpaceEx [FGD+11]. It is capable of verifying most hybrid systems by means of flowpipe-based reachability analysis using support functions. Using HyPro, a library for state set representations, we created the prototype of a reachability algorithm to put our ideas to the test.

Outline of the Thesis We will start by giving an explanation of hybrid systems and of hybrid automata, which will be used to model these systems. Afterwards, we introduce flowpipe-based reachability analysis on hybrid systems followed by an introduction of different methods to represent state sets in the reachability analysis. For the set representations, we put our focus on polytopes, support functions and ellipsoids, and discuss their advantages and disadvantages. We will then lead over to the main part of this thesis where we give an explanation of how to separate non-autonomous systems into an autonomous and a non-autonomous part. Further, we show how to use ellipsoids to reduce computation time on those separated systems. This includes an explanation of how to deal with the drawbacks of actual arithmetics, i.e. arbitrary precise numbers. Afterwards, we will further optimize this approach by reducing the computational complexity of the involved support functions. We will then give an evaluation of those optimizations based on the experimental results of our implementation for the HyPro-project[1]. At the end we will deliver a conclusion of the thesis as well as a conclusion of all results and give some ideas for further improvements.

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

CHAPTER 2

PRELIMINARIES

Before we can discuss our ideas on optimizations for the reachability analysis we have to introduce required background information for fully understanding the main part of this thesis. We will start with hybrid systems and an explanation of flowpipe-based reachability analysis followed by introducing different ways of representing sets. All set operations used in this thesis can be found in Appendix A.

2.1 Hybrid Systems

This thesis will, as stated before, deal with reachability analysis of hybrid systems. As already mentioned hybrid systems are systems with a discrete as well as a continuous behavior. A simple example for a system that can be expressed as a hybrid system is a bouncing ball. Imagine you are dropping a ball from your hand. The ball will fall towards the ground while continuously accelerating. It will then hit the ground and bounce back instantly, jump up in the air and fall down again. Looking at this example as a hybrid system we can model the airborne behavior of the ball as the continuous part of the system and the instantaneous bounce as the discrete part.
In order to precisely specify such a system, we will now take a quick look at modeling formalisms for hybrid systems and then give another example for a hybrid system. One of the most popular modeling formalism for hybrid systems are hybrid automata [GGM06, GG09], which will be used here. Aside hybrid automata, there are hybrid Petri nets and hybrid programs, which can also be used to represent hybrid systems [DA01].
The formal notion of a hybrid automaton is as follows [ACHH93].

Definition 2.1 (Hybrid automaton: Syntax)
A hybrid automaton is a tuple $\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$, where

- Loc is a finite set of *locations* or *control modes*.
- $Var = \{x_1, \ldots, x_d\}$ is a finite ordered set of real-valued *variables*, the number d is called the *dimension* of $\mathcal{H}$. $\dot{Var}$ denotes the set $\{\dot{x}_1, \ldots, \dot{x}_d\}$ of first derivatives

of the variables, and Var' the set $\{x'_1, \ldots, x'_d\}$ of values directly after a discrete change.

- $Flow : Loc \to Pred_{Var \cup \dot{Var}}$ specifies for each location its *dynamics* or *flow*.
- $Inv : Loc \to Pred_{Var}$ assigns to each location an *invariant*.
- $Edge \subseteq Loc \times Pred_{Var} \times Pred_{Var \cup Var'} \times Loc$ is a finite set of *discrete transitions* or *jumps*. For a jump $(l_1,g,r,l_2) \in Edge$, l_1 is the source location, l_2 is its target location, g specifies the jump's guard, and r its reset function, where primed variables represent the state after the jump.
- $Init : Loc \to Pred_{Var}$ assigns to each location an *initial* predicate.

Note that $Pred_X$ denotes the set of all predicates with free variables from X.

Hybrid automata are based on discrete transition systems, consisting of a set of locations, a set of variables and a set of discrete transitions (often called jumps). States (also called modes) in this system are specified by the current location paired with the current variable values. If the guard of a jump is satisfied in the current state a discrete state change can happen (we call such a transition *enabled*). The jump might change the current location as well as the values of the variables according to a reset function.
Hybrid automata extend these models by including a dynamic continuous behavior. While the control stays in a location, time transitions (called flow) let the values of the variables evolve continuously according to the dynamics of the current location, which are specified by ordinary differential equations (ODEs). Further, jumps are not urgent, i.e. jumps need not to be taken as soon as they are enabled. However, invariants for locations are used to restrict the time evolution and force the control to change the modus before the invariants get violated.

Example 2.2
As an example consider the heating system of an office, where the heater is turned on if the temperature t falls below 21°c and is turned of at over 23°c. Turning the heater on or off is a discrete action, the change of temperature over time is a continuous change in the system. Lets assume the room cools by 0.5°c each minute if the heater is off and heats up by 1°c if it is turned on. Further, we assume the temperature in the office lies always between 20°c and 24°c. The system can then be described by the hybrid automaton as shown in Figure 2.1

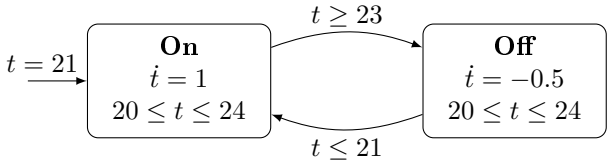


Figure 2.1: Hybrid automaton representing the behavior of a heater.

From Definition 2.1 and the behavior of a hybrid system explained before we can derive the semantics for hybrid automata.

Definition 2.3 (Hybrid automaton: Semantics)
The one-step semantics of hybrid automaton $\mathcal{H} = (Loc, Var, Flow, Inv, Edge, Init)$ of dimension d is specified by the following operational semantic rules:

$$\frac{\begin{array}{c} l \in Loc \quad v,v' \in \mathbb{R}^d \quad f : [0,\delta] \to \mathbb{R}^d \\ df/dt = \dot{f} : (0,\delta) \to \mathbb{R}^d \quad f(0) = v \quad f(\delta) = v' \\ \forall \epsilon \in (0,\delta).f(\epsilon),\dot{f}(\epsilon) \models Flow(l) \quad \forall \epsilon \in [0,\delta].f(\epsilon) \models Inv(l) \end{array}}{(l,v) \xrightarrow{\delta} (l,v')} \; Rule_{Time}$$

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

Now that we have specified hybrid automata along with their semantics, we can advance towards their reachability analysis facing the reachability problem: Given a hybrid automaton $\mathcal{H}$ and a set of states S , is there a state $s \in S$ that is reachable in $\mathcal{H}$? It can be shown that this problem in general is undecidable by reducing the hybrid automata to a 2-rate timed system[1] [ACHH93]. Fortunately, for some subclasses of hybrid systems decidability can be shown by reduction to a timed automaton. The different subclasses together with their restrictions and decidability-result can be found in Table 2.1. A more detailed explanation of those decidability results and the corresponding proofs can be found in [HKPV95].

Subclass	derivatives	conditions	bounded reach.	unbounded reach.
TA	$\dot{x} = 1$	$x \sim c$	Yes	Yes
IRA	$\dot{x} \in [c_1,c_2]$	$x \in [c_1,c_2]$ jump must reset x when $\dot{x}$ changes	Yes	Yes
RA	$\dot{x} \in [c_1,c_2]$	$x \in [c_1,c_2]$	Yes	No
LHA I	$\dot{x} = c$	$x \sim g_{linear}$	Yes	No
LHA II	$\dot{x} = f_{linear}$	$x \sim g_{linear}$	No	No
HA	$\dot{x} = f$	$x \sim g$	No	No

Table 2.1: Decidability results for subclasses of hybrid automata, where $\sim \in \{<, \leq, =, \geq, >\}$ (TA = timed automata, IRA = initialized rectangular automata, RA = rectangular automata, LHA I = hybrid automata with constant derivatives, LHA II = hybrid automata with linear ODEs, HA = general hybrid automata).

2.2 Flowpipe-Based Reachability Analysis

One major part of safety verification of hybrid systems is the computation of sets of reachable states. Safety properties specify a set of bad states, whose reachability is critical for the safety of the system. If the system never reaches such a bad state it is considered as safe, i.e. if the intersection of the reachable state set with the bad state set is empty. In general this problem is undecidable as explained in the previous

[1] 2-rate timed systems are timed systems with two different clocks. One clock progresses with rate m and one with rate n, with $m \neq n$.

chapter. The problem becomes semi-decidable if the time is discretized. However, infinite runs in the system are till possible. Thus, most approaches aim at computing an over- or under-approximation of the set of reachable states by limiting flow durations and the number of jumps (bounded reachability analysis). For the computation of the reachable sets there exists different approaches. Some reachability analysis tools for hybrid systems use approaches that are based on theorem proving [ADI03], others use SMT-solving-based approaches [GT08]. In this thesis we will focus on flowpipe-construction-based (or flowpipe-based) approaches for reachability analysis. In this approach we iteratively compute sets of states which over-approximate parts of the actual flow.

Since the actual reachability problem is undecidable we will perform a bounded reachability analysis. The bound on the flow duration is often called the *time horizon* and the bound on the number of jumps is referred to as the *jump depth*. For the rest of this thesis we will focus on over-approximations as they have more relevance in current research. However, in theory all of our approaches described in Chapter 3 also work with under-approximations as well. If the set of reachable states is over-approximated, the results of the analysis allow to declare a system safe (within the specified bounds) if the intersection with the set of bad states is empty. However, the analysis provides no conclusions if the intersection is not empty. In this case the bad states might intersect with parts of the state set introduced by over-approximation and not with the actual state set. Thus, a more precise computation is needed for further derivations.

The general algorithm for this approach is presented in Algorithm 1. The name flowpipe refers to the set of states reachable by passing time in a location which resemble a pipe due to the continuity of time.

Algorithm 1 Flowpipe-based reachability analysis

Input: Hybrid system model $\mathcal{H}$
Output: Set of reachable states $R_{\mathcal{H}}$.

1: $R \leftarrow Init_{\mathcal{H}}$
2: $R_{new} \leftarrow R$
3: **while** $R_{new} \neq \emptyset$ **do**
4: let $stateset \in R_{new}$
5: $R_{new} \leftarrow R_{new} \setminus \{stateset\}$
6: $R' \leftarrow computeFlowPipe(stateset)$
7: $R_{new} \leftarrow R_{new} \cup computeJumpSuccessor(R')$
8: $R \leftarrow R \cup R' \cup R_{new}$
9: **if** $R \cap bad\_states \neq \emptyset$ **then**
10: $break\_loop$
11: **end if**
12: **end while**
13: **return** R

The idea of the algorithm is to keep track of the set of states R, containing all states that are currently known to be reachable and R_{new}, the set of states which still need to be analyzed. First, R_{new} is initialized with the sets of all initial states (Line 2). Then, an unprocessed set *stateset* is picked from R_{new} (Line 4). Afterwards, the

set of states reachable from *stateset* by letting time pass according to the specified time horizon is computed (Line 6). For this set of states we then compute the states reachable by taking a jump transition (Line 7).
The flowpipe, i.e. the set of states reachable by time transitions, is computed by the *computeFlowPipe* method (Line 6), which returns a set $R'$of state sets that cover the flowpipe, starting in *stateset*, for a time-bounded search up to the given time horizon. The actual computation of the flowpipe will be explained later on in Chapter 2.2.1.
After the computation of the flowpipe, all possible jump successors for all state sets in R' are computed by the method *computeJumpSuccessors* (with respect to the jump depth). This involves intersecting those sets with the guards of the jumps starting in the respective location. If the intersection is not empty the reset function of the jump will be applied to compute the successor sets. Those successor sets are then collected and added to R and R' (Line 7-8).
The algorithm terminates if either a bad state is discovered to be reachable (Line 9) or R_{new} is empty (Line 3), which means all reachable states have been computed and the system is safe within the specified bounds.

2.2.1 Flowpipe-Computation

An essential part of Algorithm 1 is the computation of the flowpipe. Given an initial set, we want to compute an over-approximation of the time successors of this set within the given time horizon N. A common technique to realize this is to discretize the time horizon into equal parts, often called time steps, of length δ and then iteratively compute the set of reachable states for each time step. We will refer to these sets as *segments* of the flowpipe.

In general, the flowpipe is computed by over-approximating the first segment of the flowpipe, such that the set is covering all sets in the time interval [0,δ], and then compute all remaining segments by a recurrence relation that depends on the constant step size δ and the dynamics A in the given location. Note that δ directly influences the accuracy of the flowpipe. However, for smaller δ more segments have to be computed and thus the computation will take more time.

The following explanation of how to actually compute the flowpipe applies only to linear hybrid models[1]. For non-linear ones we refer to [Che15].

We will be covering two types of hybrid systems. Pure hybrid systems, also called *autonomous systems* and hybrid systems with some kind of external influence, called *non-autonomous systems*. Autonomous systems are systems that only depend on variables of the system. Non-autonomous systems are autonomous systems that are continuously influenced by external sources. An example for a non-autonomous system is a bouncing ball, as explained in Chapter 2.1, where its velocity is continuously influenced by the wind. Here, the original bouncing ball is the autonomous part and the wind is the non-autonomous part of the system.

The flow inside a location of an autonomous linear hybrid system is described by a

[1] Models with linear guards and derivatives as already shown in Table 2.1.

system of linear ODEs [GGM06, GG09, KV11] of the form

$$\dot{X}(t) = A \cdot X(t), \tag{2.1}$$

where $X(t)$ is the state set at time t and A is a matrix describing the flow.
Non-autonomous systems extend these dynamics by a time-dependent m-dimensional interval vector $u(t)$, used to represent the external influences on the systems evolution. This influence results in a dynamic behavior that can be described by

$$\dot{X}(t) = A \cdot X(t) + B \cdot u(t) = A \cdot X(t) + V(t), \tag{2.2}$$

where $B \in \mathbb{R}^{d \times m}$, $u(t) \in \mathbb{R}^m$ and $V(t) = B \cdot u(t)$ [GGM06].

Solving Equation 2.1 for autonomous systems results in

$$X(t) = \mathrm{e}^{tA} X_0.$$

This formula specifies the states reachable from the initial state X_0 with the flow matrix A at time t. Using this and the time discretization the set of states reachable from a state set X_i with a time step δ can be obtained by the recurrence relation

$$X_{i+1} = \mathrm{e}^{\delta A} X_i = \Phi \cdot X_i.$$

This relation allows us to iteratively compute the set of reachable states for a certain segment by applying a linear transformation Φ on the previously computed segments.

Solving Equation 2.2 for non-autonomous systems would result in

$$X(t) = \mathrm{e}^{tA} X_0 + \int_0^t \mathrm{e}^{(t-s)A} V(s)\ ds. \tag{2.3}$$

Solving this integral is quite expensive. Thus, this part is commonly over-approximated by a set representing a ball with radius β [GGM06], where β is an over-approximation of the maximum of the integral. For a given norm $\|\cdot\|$ the set is given by

$$\mathcal{B}_\beta = \{x \in \mathbb{R}^n \mid \|x\| \leq \beta\}.$$

So far we only know how to compute the reachable segments for times $i \cdot \delta$. In order to obtain an actual approximation of the flowpipe we need to cover the behavior in between the segments as well. If we want to be able to make statements about the reachable states for a time interval $[i \cdot \delta, (i+1) \cdot \delta]$ an over-approximation has to be computed. It can be shown that it is sufficient to compute the over-approximation for the interval $[0,\delta]$ and then use this set for the iterative computation. Due to the systems time invariance we can ensure that the continuous behavior in all further segments is covered by this computation [GG09].
Currently there are two main approaches used to compute the first segment, both using bloating[1] to completely cover the flowpipe within the time interval $[0,\delta]$.

We refer to the first approach as "*uniform bloating*" and to the second method as "*improved/selected bloating*".

[1] The phrase *bloating* denotes an enlargement of a given set often realized by a Minkowski sum with another set Ω, where the origin is an element of Ω.

Uniform Bloating [GGM06, GG09] covers the dynamics of the autonomous part by adding a bloating factor α to the convex hull of the union of the initial set X_0 and $e^{\delta A}X_0$. For non-autonomous systems, an additional bloating factor β can be computed to cover the influences of external inputs. Using this method, the first segment Ω_0 can then be computed as

$$\Omega_0 = conv(X_0 \cup e^{\delta A}X_0) \oplus \mathcal{B}_{\alpha+\beta},$$

where $\mathcal{B}_{\alpha+\beta}$ is a ball of radius $\alpha + \beta$. The factor α depends on the shape of the flow A and the step size δ [GGM06, Gir05]. It is over-approximation of the deviation from the trajectory between two points and the line connecting those two points. This over-approximation then describes the deviation of the flow from the convex hull and thus the bloating with factor α is necessary to cover this deviation of the flow between the two segments. Further, the factor β is needed to cover the non-autonomous part of the system (if the system is fully autonomous β should be equal to zero and thus can be neglected). One way to compute the parameters α and β that ensures all continuous behavior is covered is given in Lemma 2.4 [GGM06, Gir05].

Lemma 2.4
Given a norm $\|\cdot\|$, let $\mu = \sup_{u\in U}\|u\|$ and $\nu = \sup_{x\in X_0}\|x\|$ then:

$$\Omega_0 = CH(X_0 \cup e^{\delta A}X_0) \oplus \mathcal{B}_{\alpha_\delta+\beta_\delta},$$

where $\alpha_\delta = (e^{\delta\|A\|-1} - \delta\|A\|)\nu$, $\beta_\delta = \frac{e^{\delta\|A\|-1}}{\|A\|}\mu$ and $\mathcal{U}$ is the state set specifying the external input.

In order to improve accuracy one could replace the bloating with a ball by a bloating with a more accurate representation, for example using boxes or polytopes where the set is then created not via β but the actual solution of the differential equation from Equation 2.3. For for most scenarios however a ball is sufficient as the necessary bloating and the external input are in general small compared to the actual segments and thus the representation of the external input has but a minimal influence on the reachable sets.

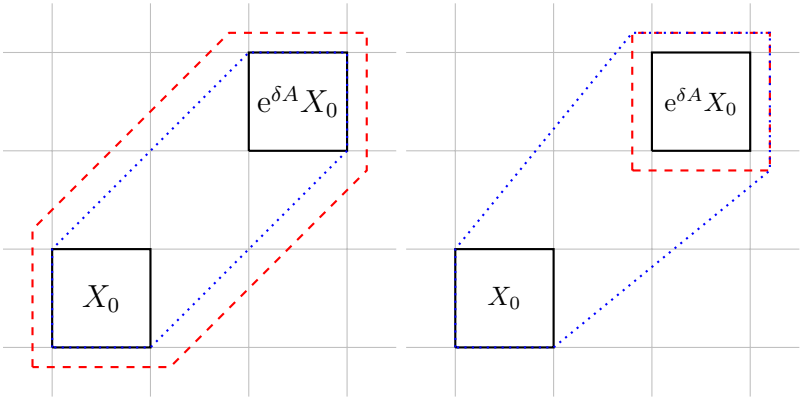
In contrast to the uniform bloating, which first computes the convex hull of the union of two sets and applies bloating afterwards, the *improved bloating* approach bloats the reachable set at time δ and computes the convex hull of the union of the initial set X_0 and the bloated set at time δ [Gir05, GGM06]. The first segment is then computed as

$$\Omega_0 = conv(X_0 \cup (e^{\delta A}X_0 \oplus \mathcal{B}_{\alpha+\beta})).$$

A graphical comparison of those two approaches can be seen in Figure 2.2. As seen in the illustration, the improved bloating, as the name already suggests, is a more accurate over-approximation reducing the size of the resulting set.
Once the first segment is computed, all further segments up to the time horizon can be computed iteratively by applying the linear transformation Φ to the previously obtained segment. These segments Ω_i, computed by

$$\Omega_i = \Phi\Omega_{i-1},$$

each cover the time interval $[i\delta,(i+1)\delta]$. Thus, $\bigcup_i \Omega_i$ with $i \in \{1,\ldots,\lceil\frac{N}{\delta}\rceil\}$ cover the whole flowpipe within the time horizon N.



(a) Uniform bloating (b) Improved Bloating

Figure 2.2: Illustration of the uniform bloating approach and the improved bloating approach. Dashed lines represent the bloating and dotted lines the convex hull.

Example 2.5
Figure 2.3 shows an illustration of a possible flowpipe approximation, where the initial set Ω_0 is computed using uniform bloating.

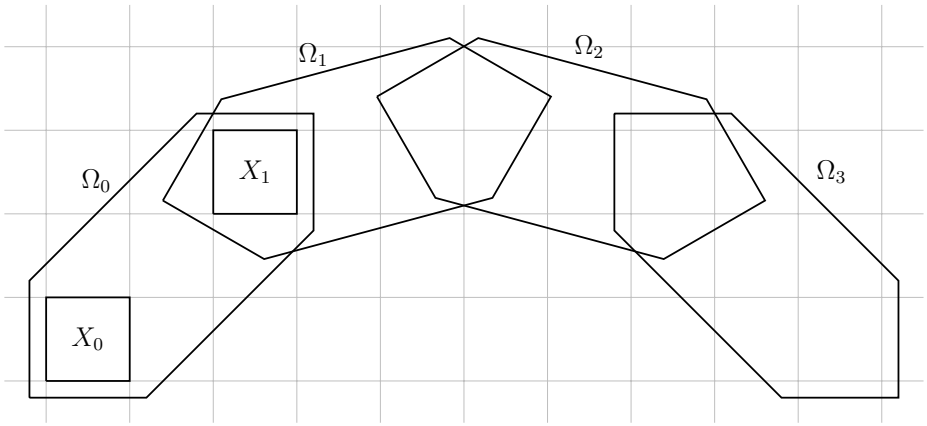


Figure 2.3: Flowpipe computation as an over-approximation of the reachable set.

For autonomous systems only the computation of Ω_0 involves bloating. However, for non-autonomous systems an additional bloating is needed after each step to cover the continuous influence of the external input. This results in an alternating application of linear transformation and Minkowski sum, which in strongly increases the complexity of the flowpipe computation. The corresponding sets are computed by

$$\Omega_{i+1} = (\Phi\Omega_i) \oplus \mathcal{B}_\beta.$$

An alternative way of computing those non-autonomous flowpipes which allows to separate these operations and use further optimizations will be shown later on in

Chapter 3.

As an example for the influence that the external input has on the reachability analysis can be seen in Figure 2.4. Here the blue dashed sets represent the flowpipe segments without external influence. The continuous growth of the segments introduced by the external influence is clearly visible. Comparing the set Ω_3 with the corresponding autonomous set we see that the size of the set has almost doubled even though the external input has only a small influence on Ω_0. Even though in general the external influence is smaller than illustrated here, it still can have a huge impact on flowpipes with many segments.

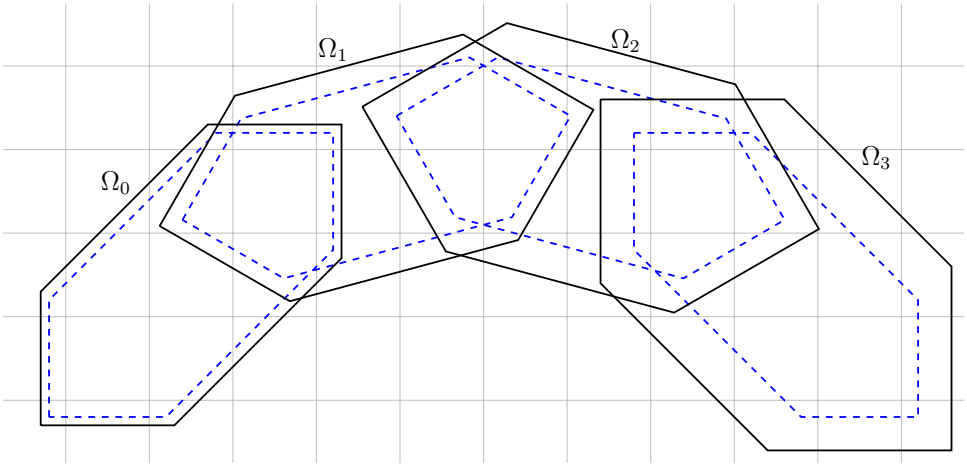


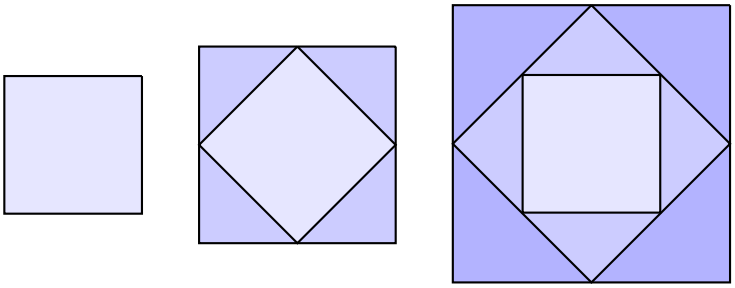
Figure 2.4: Comparison of the flowpipes with and without external influences.

2.2.2 Wrapping Effect

One of the problem that most common ways to compute reachability face, as soon as approximation is involved, is the wrapping effect. This also holds for the flowpipe-based approach described in Chapter 2.2.1. All reachability algorithms use over- or under-approximation during the computation of a flowpipe to ensure the decidability of the reachability problem. This may lead to an approximation error for current segments which also influences all further segments. In Figure 2.5 we can see a box[1] being turned by 45° and then over-approximated by a box twice. Without over-approximation the size would not increase but with over-approximation its size has grown to more than three times its original size after just two transformations. One reason for the wrapping effect to occur is that we have to ensure the given representation is closed under all operations. Further, the wrapping effect affects all convex representations. Even though, for some representations the wrapping effect can appear on different operations. For boxes the effect appears on linear transformation, but not on Minkowski sum and for ellipsoids it appears on Minkowski sums but not on linear transformations. However, for all convex sets the wrapping effect appears

[1] A box is described by its lower and upper bound in each dimension and can thus be represented by an interval-vector.

on union operations.



(a) Ω_0 (b) Ω_1 (c) Ω_2

Figure 2.5: Successive wrapping induced by a rotation of $\frac{\pi}{4}$ of a box representation with over-approximation to ensure that boxes are closed under linear transformation. It holds $\Omega_1 = approx(\Phi\Omega_0)$ and $\Omega_2(approx(\Phi\Omega_1))$ where the darkest parts describe the over-approximation.

Taking a look back at the flowpipe construction, especially for the non-autonomous systems, we often obtain long sequences of operations containing many linear transformations and Minkowski sums. Linear transformations and Minkowski sums often use over-approximations and thus may yield a wrapping effect which can grow further with each upcoming operation. As a result, reachability analysis becomes more and more inaccurate the longer the flowpipes get and the more jumps occur. Of course, this effect can be minimized by choosing a representation that does not use approximations on those two operations like support functions or zonotopes. However, for those representations the wrapping effect occurs in other parts of the analysis. Due to this wrapping effect the intersection of bad states with the computed flowpipe might be non-empty whereas the intersection of the actual set without wrapping could be empty. Note that the total error introduced by the wrapping effect scales with the size of the set which is approximated. The approach introduced in Chapter 3 will minimize the wrapping effect by reducing the size of sets that need to be approximated.

2.3 Set Representation

As stated in Chapter 2.2.1 it is impossible in most cases to compute the exact reachable set. In order to be able to obtain results in a reasonable time we need to approximate the actual reachable set by a set for which the reachability analysis is fast and still precise. The decision between computing sets fast or precise and the different characteristics of hybrid systems, e.g. a high number of jumps or long flowpipes, gave rise to many set representations which each perform well on some areas but none of the representations is superior on all areas.

At the current state of the art there exists many established ways to represent a set. One could use boxes, ellipsoids, polytopes, zonotopes or support functions just to name a few possibilities. Each of these representations has its own positive and negative characteristics. For example, if we use support functions we are able to compute

Minkowski sums in constant time but the intersection of support functions is quite complex, whereas the intersection of $\mathcal{H}$-polytopes is very easy to compute but the Minkowski sum is hard to compute. In Table 2.2 you can find a comparison of the most common representations and how they behave for the different operations. It

	Size	$A\cdot$	$\cdot \oplus \cdot$	$CH(\cdot \cup \cdot)$	$\cdot \cap \cdot$
Boxes	$2d$	+	+	+	+
Ellipsoids	$d^2 + d$	+	+	+	-
$\mathcal{H}$-Polytopes	$kd + k$	+*	-	-	+
$\mathcal{V}$-Polytopes	kd	+	+	+	-
Zonotopes	$kd + d$	+	+	-	-
Support Functions	NA	+	+	+	-

Table 2.2: A comparison of different operations on different representations. + menas easy to compute, - means hard to compute. *only if A is invertible

can be shown that each representation has its benefits and will for some scenario be a better fit than the others. Also, all those representations are not equally good in terms of accuracy. Support functions are the only representation that can represent all convex objects without the need of approximation (other representations are not capable of describing bended surfaces of objects). Furthermore, some representations are able to represent a body with an arbitrary but finite number of facets[1], like support functions and polytopes. Zonotopes are able to represent point-symmetric sets leaving boxes to be the most coarse representation in the list but also the easiest to compute. Boxes are well fitted for a quick but unprecise approximation of reachable states.

In the following, we are going to look at polytopes, support functions and ellipsoids in more detail.

2.3.1 Polytopes

The class of polytopes is actually divided into two different representations, $\mathcal{H}$-polytopes and $\mathcal{V}$-polytopes. However, it is possible to convert between those two representations even though this conversion is hard to compute. We will start with an explanation of $\mathcal{H}$-polytopes, then we will deal with $\mathcal{V}$-polytopes and afterwards explain the conversion between those two.

Definition 2.6 (Closed halfspace)
A d-dimensional closed halfspace is a set $\mathrm{H} = \{x \in \mathbb{R}^d \mid c^T x \leq z\}$.
The supporting hyperplane for H is given by $\mathcal{Z} = \{x \in \mathbb{R}^d \mid c^T x = z\}$.

Definition 2.7 ($\mathcal{H}$-polytope)
A d-dimensional $\mathcal{H}$-polyhedron $\mathcal{H} = \bigcup_{i=1}^{n} \mathrm{H}_i = \bigcap_{i=1}^{n} \{x \in \mathbb{R}^d \mid c_i \cdot x \leq z_i\}$ is the intersection of finitely many closed halfspaces. A bounded $\mathcal{H}$-polyhedron is called a $\mathcal{H}$-polytope.

In other words a $\mathcal{H}$-polytope is a collection of points satisfying the intersection of linear inequalities and is therefore often denoted $\mathcal{H} = \{x \in \mathbb{R}^d \mid Cx \leq z\}$. A simple

[1] Facets are flat faces of objects.

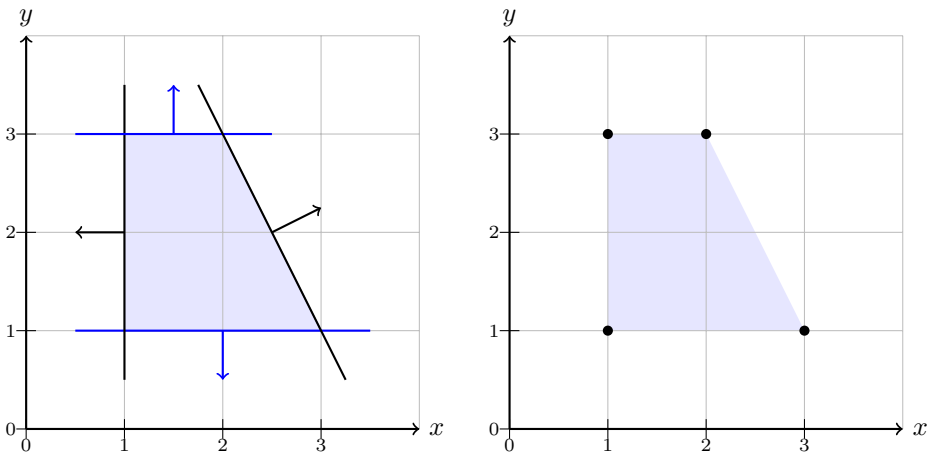
example of a $\mathcal{H}$-polytope can be seen in Figure 2.6(a). It arises from the the following constraints.

$$\begin{array}{rcrcr} & & y & \leq & 3 \\ & - & y & \leq & -1 \\ -x & & & \leq & 1 \\ x & + & 0.5y & \leq & 3.5 \end{array}$$

Next, we take a look at $\mathcal{V}$-Polytopes. $\mathcal{V}$-Polytopes are not defined using the intersection of halfspaces but a set of points in the d-dimensional space.

Definition 2.8 ($\mathcal{V}$-polytope)
A $\mathcal{V}-$polytope $\Omega = CH(V)$ is the convex hull of a finite set of points $V \subset \mathbb{R}^d$.

Again, a simple example of a $\mathcal{V}$-polytope can be seen in Figure 2.6(b) which consists of a collection of points and the corresponding convex hull.



(a) $\mathcal{H}$-polytope (b) $\mathcal{V}$-polytope

Figure 2.6: A $\mathcal{H}$-polytope and a $\mathcal{V}$-polytope representing the same set.

After defining $\mathcal{V}$-polytopes and $\mathcal{H}$-polytopes we can take a look at the different operations on those sets. One major property that is exploited in the computation is that we can convert between both representations and thus can use the easier to compute version of all operations.
For Minkowski sums and convex unions we will use $\mathcal{V}$-polytopes and intersections will be performed on $\mathcal{H}$-polytopes. The only operation that can be efficiently computed on both sets is the linear transformation (If the linear transformation is invertible).

Example 2.9
Assume we are given two $\mathcal{H}$-polytopes $\mathcal{H}_1$,$\mathcal{H}_2$ and want to compute the Minkowski sum of those polytopes. Instead of computing $\mathcal{H}_1 \oplus \mathcal{H}_2$ we would use the more efficient computation

$$convTo\mathcal{H}(\ convTo\mathcal{V}(\mathcal{H}_1) \oplus convTo\mathcal{V}(\mathcal{H}_2)\ ),$$

where $convTo\mathcal{H}$ and $convTo\mathcal{V}$ return a converted polytope in $\mathcal{H}$ and $\mathcal{V}$ representation respectively.

Now that we stated which operation is best computed on which polytope, we are going to explain how these operations are actually computed on polytopes.

We will begin with the intersection of two $\mathcal{H}$-polytopes $\mathcal{H}_i = \{x \mid C_i x \leq z_i\}$, $i \in \{1,2\}$. As both are represented by the intersection of inequalities we can obtain the intersection by combining the set of inequalities and get the new $\mathcal{H}$-polytope

$$\mathcal{H}_1 \cap \mathcal{H}_2 = \{x \in \mathbb{R}^d \mid \begin{smallmatrix} C_1 \\ C_2 \end{smallmatrix} x \leq \begin{smallmatrix} z_1 \\ z_2 \end{smallmatrix}\},$$

where redundant constraints may occur. However, removing those redundant constraints is not mandatory as they do not affect the resulting sets but the amount of memory necessary to store the set.

Next, we have the Minkowski sum of two $\mathcal{V}$ polytopes $\mathcal{V}_1$,$\mathcal{V}_2$, which is computed as

$$\mathcal{V}_1 \oplus \mathcal{V}_2 = \{x + y \mid x \in \mathcal{V}_1, y \in \mathcal{V}_2\}.$$

The convex union of $\mathcal{V}$-polytopes $\mathcal{V}_1$,$\mathcal{V}_2$ is computed as the convex hull of the two sets of points V_1, V_2

$$CH(\Omega_1 \cup \Omega_2) = conv(\{V_1 \cup V_2\}).$$

The only operation not covered now is the linear transformation. Lets start with the linear transformation of a $\mathcal{V}$-polytope $\mathcal{V}$ with transformation matrix Φ. $\Phi\mathcal{V}$ can be computed by

$$\Phi\mathcal{V} = \{\Phi \cdot x \mid x \in \mathcal{V}\}$$

Due to the properties of linear transformation each extreme-point of $\mathcal{V}$ is also a extreme-point[1] of $\mathcal{V}'$. Thus, it would suffice to only compute the linear transformation of those points. For a direct linear transformation on $\mathcal{H}$-polytopes we need to ensure that Φ is invertible. If it is not, we would need to convert to a $\mathcal{V}$-polytope in order to perform the linear transformation. The linear transformation on a $\mathcal{H}$-polytope is performed on each of its halfspaces $\mathcal{H}_i$ separately.
In order to apply the linear transformation to a halfspace H we have to choose d arbitrary but linear independent points from its supporting hyperplane $\mathcal{Z}$. The linear transformation Φ will then be applied to those d points. Afterwards, we compute the hyperplane $\mathcal{Z}'$ on which the linear transformed points lie. In order to generate the resulting halfspace $\mathrm{H}' = \Phi\mathrm{H}$ supported by $\mathcal{Z}'$ we need to transform a point $x \in \mathrm{H} \setminus \mathcal{Z}$ and choose the orientation of H such that $\Phi x \in \mathrm{H}'$.
Now that we have considered all operations for both representations we can discuss the conversion between the representations.

The conversion of a $\mathcal{H}$-polytope into a $\mathcal{V}$-polytope is often referred to as the *facet enumeration problem*. The conversion of a $\mathcal{V}$-polytope into a $\mathcal{H}$-polytope is called the *vertex enumeration problem*. Both problems are of great interest, even outside of the context of reachability analysis. Also, there are good algorithms for both problems if only 2 or 3 dimensions are involved. For higher dimensions there are also algorithms but their complexity is much higher. There is the class of beneath-beyond-methods dealing with the conversion from $\mathcal{V}$ to $\mathcal{H}$ [BDH96, Bor07] and the class of reverse search algorithms dealing with both directions of the conversion [KA96].

[1] A point that can not be removed from the $\mathcal{V}$-polytope without reducing its size.

For the conversion from a d-dimensional $\mathcal{H}$-polytope to a $\mathcal{V}$-polytope a naive way would be to compute all points where d of the halfspaces of $\mathcal{H}$ intersect and then test if the point is an element of the remaining halfspaces. This way the only points that will be left are exactly the extreme-points of the polytope. The other direction of the conversion can be done by computing all halfspaces generated by the sets of points $V_i \subseteq V$, with $|V_i| = d$ and testing if all points of $\mathcal{V}$ are elements of the halfspace. The resulting halfspaces then generate the corresponding $\mathcal{H}$-polytope. The resulting $\mathcal{H}$-polytope will be free of be redundant halfspaces, thus the conversion from $\mathcal{H}$ to $\mathcal{V}$ and back to $\mathcal{H}$ could be used in order to remove redundant hyperplanes.

2.3.2 Support Functions

As the name already suggests we can also represent a set by a function called a *support function* (short sf). This support function assigns to each direction vector l a distance value such that the point $l \cdot \rho_\Omega(l)$ lies on the hyperplane $\mathcal{Z}_l$ with normal l. $\mathcal{Z}_l$ then also touches Ω and all elements of Ω lie within the halfspace H induced by $\mathcal{Z}_l$.

Definition 2.10
The Support function of a set Ω, denoted ρ_Ω is defined by:

$$\begin{aligned} \rho_\Omega : \mathbb{R}^d &\longrightarrow \mathbb{R} \cup \{-\infty,\infty\} \\ l &\longmapsto \sup_{x \in \Omega} x \cdot l \end{aligned}$$

A point x of Ω with $x \cdot l = \rho_\Omega(l)$ is called a support vector of Ω in direction l.

Example 2.11
An illustration of the connections between the support value, the vector l, the corresponding hyperplane $\mathcal{Z}_l$ and the induced halfspace H can be seen in Figure 2.7.

One thing to know about support functions is that they are more expressive than most other representations as the expressed set can contain curves. It can be shown that a finite $\mathcal{H}$-polytope can never express a set representing a ball, which of course is can be done with support functions. It is also possible and almost straight forward to convert the most common representations into support functions. The connection between different representations and the corresponding support functions for the most commonly used representations is given below [GG09].

- The unit ball for the usual Euclidean norm: $B_2 = \{x \in \mathbb{R}^d : \|x\|_2 \leq 1\}$. Then,

$$\rho_\Omega(l) = \|l\|_2$$

- The unit ball for the ∞-norm: $B_\infty = \{x \in \mathbb{R}^d : \|x\|_\infty \leq 1\}$. Then,

$$\rho_\Omega(l) = \|l\|_1$$

- An ellipsoid $\Omega = \{x \in \mathbb{R}^d : x^T Q^{-1} x \leq 1\}$. Then,

$$\rho_\Omega(l) = \sqrt{l^T Q l}$$

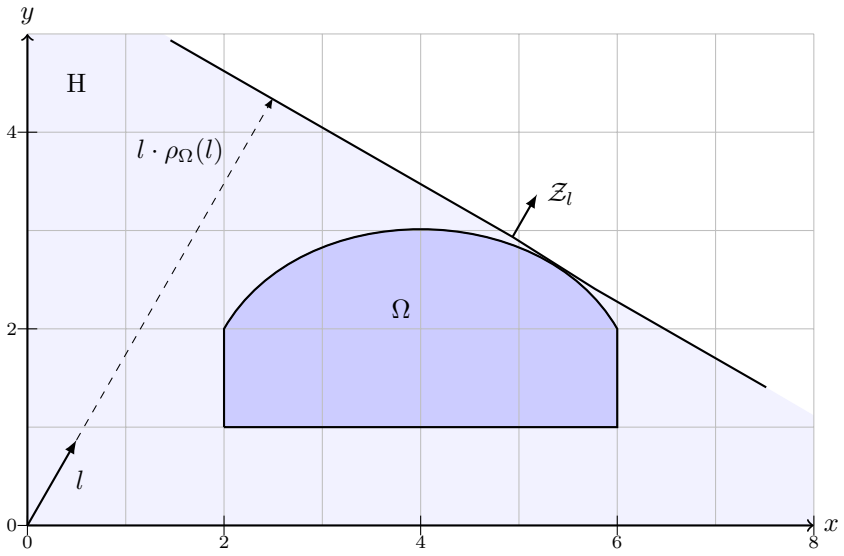


Figure 2.7: Illustration of the connection between the vector l, its support function value $\rho_\Omega(l)$ and the corresponding halfspace H_l.

- A hyper-rectangle: $\Omega = [-h_1; h_1] \times \cdots \times [-h_d; h_d]$ where $h_1, \ldots, h_d \in \mathbb{R}^+$. Then,

$$\rho_\Omega(l) = \sum_{j=1}^{d} |h_j l_j|$$

- A zonotope[1]: $\Omega = \{\alpha_1 g_1 + \cdots + \alpha_r g_r : \alpha_j \in [-1,1], j = 1, \ldots, r\}$ where the generators $g_1, \ldots, g_r \in \mathbb{R}^d$. Then,

$$\rho_\Omega(l) = \sum_{j=1}^{r} |g_j l|$$

- A polytope: $\Omega = \{x \in \mathbb{R}^d : Cx \leq d\}$ where C is a matrix and d a vector of compatible dimension. Then, computing $\rho_\Omega(l)$ is equivalent to solving a linear program:

$$\begin{cases} Maximize\ l \cdot x \\ Subject\ to\ Cx \leq d \end{cases}$$

Next, we will explain how to compute operations on support functions [GK98, GGM06, GG09]. Let us assume that ρ_{Ω_1} and ρ_{Ω_2} are the support functions of the convex sets Ω_1 and Ω_2. The convex union of those sets can then be described by

$$CH(\rho_{\Omega_1}(l) \cup \rho_{\Omega_2}(l)) = \max\{\rho_{\Omega_1}(l), \rho_{\Omega_2}(l)\}$$

for every $l \in \mathbb{R}^d$. It can be shown that the resulting function again is a support function describing the convex union.

[1] A zonotope $\mathcal{Z}$ is a d-dimensional set represented by an arbitrary number of generator vectors v_i and its center point c such that $\mathcal{Z} = \{x \in \mathbb{R}^d \mid x = c + \sum_i \lambda_i \cdot v_i,\ -1 \leq \lambda_i \leq 1, \textit{for all } i\}$.

The intersection is similarly defined as

$$\rho_{\Omega_1}(l) \cap \rho_{\Omega_2}(l) = \min\{\rho_{\Omega_1}(l), \rho_{\Omega_2}(l)\}$$

for every $l \in \mathbb{R}^d$. Again, it is not difficult to show that the min-operation performs the intersection operation. Unlike the previous case, $\min\{\rho_{\Omega_1}(l), \rho_{\Omega_2}(l)\}$ is not a support function as support values can be computed even if the set is empty or degenerated [GK98]. As a result some of the supporting halfspaces defined by this operation may be redundant. To get the corresponding support function we would need to intersect all of these halfspaces and then convert the resulting $\mathcal{H}$-polytope into a support function. Due to this problem the intersection of support functions is hard to compute. Note that the intersection of a support function with a halfspace is again a support function if the intersection is not empty.

The Minkowski sum can be efficiently realized for support functions by

$$\rho_{\Omega_1} \oplus \rho_{\Omega_2} = \rho_{\Omega_1}(l) + \rho_{\Omega_2}(l)$$

for every $l \in \mathbb{R}^d$. Note that it is not necessary to respect points of the set here but only the function. Thus, the Minkowski sum can be computed efficiently and is closed under support functions.

Last, we have the linear transformation. It can be shown that for support functions a linear transformation Φ of a set Ω is equivalent as applying the transposed matrix Φ^T to the evaluation direction.

$$\rho_{\Phi\Omega}(l) = \rho_{\Omega}(\Phi^T l)$$

As already shown at the begin of Chapter 2.3 in Table 2.2, all operations except for intersection with another support function can be efficiently computed. Therefore, support functions are well fitted for flowpipe-based reachability analysis. However, due to the way operations on support functions are computed it is not possible to give a general estimation of the storage complexity for support functions.

Reachability Analysis using support functions

Reachability analysis using support functions differs from other representations due to the way support functions are implemented in software. Unlike all other representations we do not need to compute the resulting support function after each operation. Instead, we build up a tree of operations and only evaluate the support functions with respect to the operations in the tree if needed. This improves the overall computational time for support functions.
The evaluation of such a tree of support functions is done recursively by handing the direction of interest l down from the root to the leaves of the tree.

Example 2.12
An example of a support function tree can be seen in Figure 2.8. The shown tree could have been created by first applying a linear transformation to a polytope, then computing the Minkowski sum of two balls and finally compute the Minkowski sum of those two support functions.

Assume we now want to evaluate the support function from Figure 2.8 in direction l. This would result in the following computation

$$\begin{aligned}\rho_{\Omega 1}(l) &= \max\left(\rho_{\Omega 2}(l),\, \rho_{\Omega 3}(l)\right)\\ &= \max\left(\Phi \cdot \rho_{\Omega 4}(l),\, \rho_{\Omega 3}(l)\right)\\ &= \max\left(\rho_{\Omega 4}(\Phi^T l),\, \rho_{\Omega 3}(l)\right)\\ &= \max\left(\rho_{\Omega 4}(\Phi^T l),\, \max(\rho_{\Omega 5}(l),\, \rho_{\Omega 6}(l))\right)\end{aligned}$$

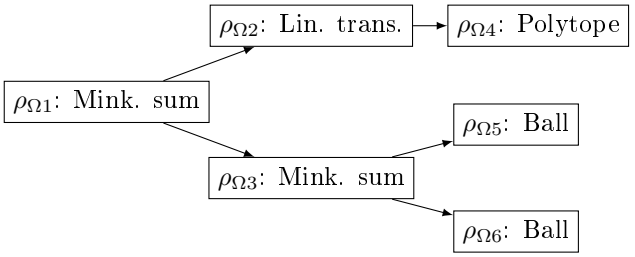


Figure 2.8: Representations of an operation tree of a support functions.

2.3.3 Ellipsoids

An uncommon way to represent sets is by means of an ellipsoid. Ellipsoids are mostly avoided in flowpipe-based reachability analysis as it has been shown that the computations of intersection as well as the union on ellipsoids are very complex and the results are unprecise due to the involved approximations needed to ensure that those operations are closed under ellipsoids. However, ellipsoids can be used to compute linear transformations and Minkowski sums very efficiently. In our approach we will be using ellipsoids only in the presence of linear transformations and Minkowski sums, thus they are well suited for this approach.

As stated before, the intersection and union of ellipsoids are complex and not needed for our purpose as we will use them to compute sets that involve neither of those operations. Therefore, for ellipsoids the intersection and union will not be covered in this thesis. For more information on intersection and union we refer to [KV07].

An ellipsoid $\mathcal{E}$ is represented by $\mathcal{E} = (q,Q)$ where q is a vector pointing to the center of $\mathcal{E}$ and Q is the shape matrix of $\mathcal{E}$. Q can be seen as a linear transformation that has to be applied to an unit ball in order to obtain the corresponding ellipsoid.

Definition 2.13 (Ellipsoid)
An ellipsoid $\mathcal{E}(q,Q)$ in $\mathbb{R}^d$ with center q and shape matrix Q is a set

$$\mathcal{E}(q,Q) = \{x \in \mathbb{R}^d \mid \langle l,x\rangle \leq \langle l,q\rangle + \langle l,Ql\rangle^{\frac{1}{2}} \textit{ for all } l \in \mathbb{R}^d\},$$

where Q is positive semi-definite ($Q = Q^T$ and $\langle x,Qx\rangle \geq 0$ for all $x \in \mathbb{R}^d$) and $\langle x,y\rangle$ denotes the scalar product of vectors x and y.

Next, we take a look at linear transformations and Minkowski sums of ellipsoids.
Let $\mathcal{E}(q,Q) \subseteq \mathbb{R}^d$ be an ellipsoid and $\Phi \in \mathbb{R}^{d\times d}$ a matrix, then the linear transformation of $\mathcal{E}$ with Φ is defined as

$$\Phi\mathcal{E}(q,Q) = \mathcal{E}(\Phi q, \Phi Q \Phi^T).$$

As the matrix $Q' = \Phi Q \Phi^T$ is again semi-definite, the result of a linear transformation as presented yields a shape matrix of an ellipsoid. Thus, ellipsoids are closed under linear transformation.

Now, for the Minkowski sums of ellipsoids. To ensure that ellipsoids are closed under Minkowski sum we need to over-approximate the actual set that is obtained by applying the Minkowski sum. An illustration of the actual Minkowski sum of two ellipsoids and an ellipsoid that over-approximates the Minkowski sum can be found in Figure 2.9

Consider the Minkowski sum of arbitrary many non-degenerated ellipsoids $\mathcal{E}(q_1,Q_1), \cdots, \mathcal{E}(q_k,Q_k)$. The resulting set is not always an ellipsoid, as shown in Figure 2.9. But, it can be tightly approximated by an external ellipsoid, i.e. we can find an ellipsoid that completely contains the set resulting of a Minkowski sum [KV07].

Let $l \in \mathbb{R}^d$ be a non-zero vector. The external approximation $\mathcal{E}(q^+,Q_l^+)$ of the sum $\mathcal{E}(q_1,Q_1) \oplus \cdots \oplus \mathcal{E}(q_k,Q_k)$ is tight in direction l, i.e.,

$$\mathcal{E}(q_1,Q_1) \oplus \cdots \oplus \mathcal{E}(q_k,Q_k) \subseteq \mathcal{E}(q^+,Q_l^+)$$

and

$$\rho(\pm l \mid \mathcal{E}(q^+,Q_l^+)) = \rho(\pm l \mid \mathcal{E}(q_1,Q_1) \oplus \cdots \oplus \mathcal{E}(q_k,Q_k)),$$

where $\rho(x \mid \mathcal{E}(q,Q)) = \langle x,q\rangle + \langle x,Qx\rangle^{\frac{1}{2}}$ is the support function corresponding to the ellipsoid $\mathcal{E}(q,Q)$ and $q^+ = q_1 + \ldots + q_k$.
It can be shown that the shape matrix Q_l^+ can be computed by

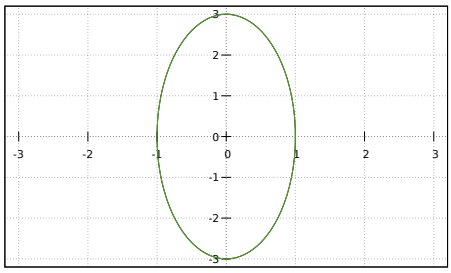
$$Q_l^+ = \left(\sum_{i=1}^{k} \langle l,Q_i l\rangle^{\frac{1}{2}}\right)\left(\sum_{i=1}^{k} \frac{1}{\langle l,Q_i l\rangle^{\frac{1}{2}}} Q_i\right).$$

This covers all necessary operations on ellipsoids that will be used later on in Chapter 3.1.

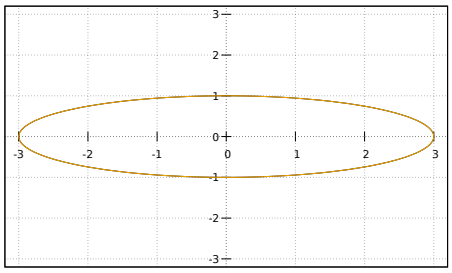
From ellipsoid to other representations

Our approach of optimizing the reachability analysis for non-autonomous hybrid systems works on most representations and thus we need to be able to convert an ellipsoid to the most common representations. Those conversions are needed as we will often have to compute the Minkowski sums of an ellipsoid and a set in another representation, where it is more efficient to first transform the ellipsoid into the other representation.

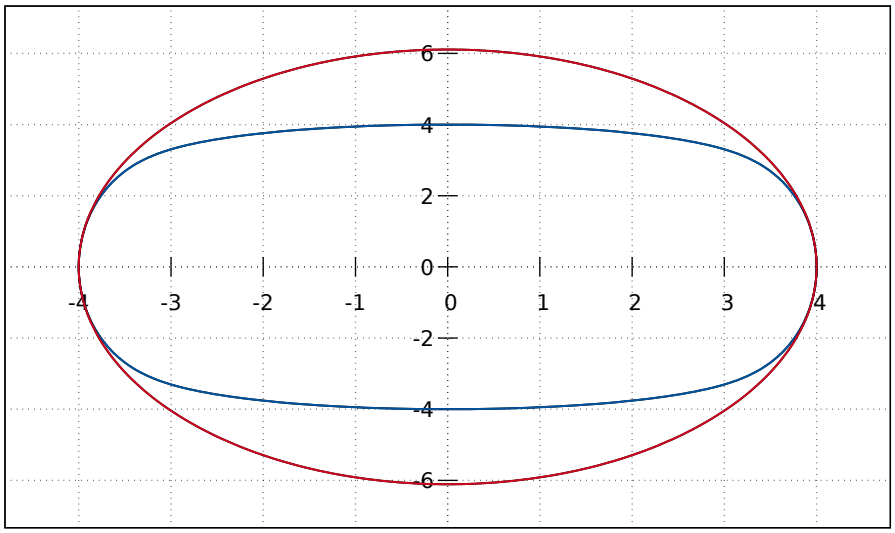
We will give a description of this conversion to support functions, $\mathcal{V}$-polytopes, $\mathcal{H}$-polytopes, boxes and zonotopes starting with the most simple conversion.



(a) First ellipsoid $\mathcal{E}_1$



(b) Second ellipsoid $\mathcal{E}_2$



(c) Minkowski sums and ellipsoidal approximation

Figure 2.9: Actual Minkowski sum (blue) and an approximating ellipsoid (red) for $\mathcal{E}_1 \oplus \mathcal{E}_2$

In order to transform an ellipsoid $\mathcal{E}(q,Q)$ to a support function we can just use the underlying support function used to evaluate ellipsoids

$$\rho_{\mathcal{E}(q,Q)}(l) = \rho(\pm l \mid \mathcal{E}(q,Q)) = \langle l,q\rangle + \sqrt{\langle l,Ql\rangle}.$$

Next, we will convert an ellipsoid into a box. Boxes are defined by an interval for each dimension and thus we can over-approximate the ellipsoid by a box $\mathcal{B}$ by evaluating $\mathcal{E}$ once in direction of each axis and obtain the bounds of the corresponding box.

$$\mathcal{B} = \{x \in \mathbb{R}^d \mid x(i) \in [q(i) - \rho_{\mathcal{E}(0,Q)}(\mathbb{1}_i); q(i) + \rho_{\mathcal{E}(0,Q)}(\mathbb{1}_i)] \forall 1 \leq i \leq d\}$$

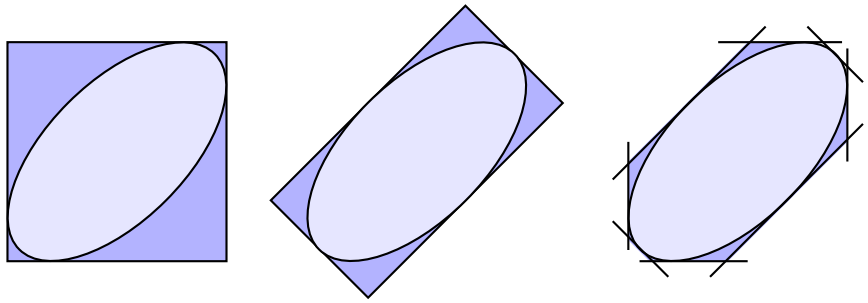
where $q(i)$ refers to q's value for the i-th dimension and $\mathbb{1}_i$ denotes the vector containing a 1 at its i-th entry and 0 for all other entries.

In order to obtain a $\mathcal{H}$-polytope from an ellipsoid we can simply evaluate it in a fixed number of directions where this number has a direct impact on the precision. For a good approximation we advise to evaluate in at least 8 uniformly distributed directions for each pair of dimensions.

Lastly, we can convert an ellipsoid into a zonotope by solving a linear optimization problem in order to obtain minimal and maximal axis for each pair of dimension and use these results as our generators of the zonotope. However, this is not an optimal solution but a fast one to compute.

In Figure 2.10 we have illustrated the results of the conversion of the same ellipsoid into different representations. As you can see, there is some approximation error introduced in the process (dark blue area) which is present for all representations but support functions.

In conclusion, ellipsoids have a good approximation only for support function and $\mathcal{H}$-polytopes. Therefore, it is advised to use ellipsoids only in the context of support functions or polytopes to reduce the approximation error.



(a) Box approx. (b) Zonotope approx. (c) $\mathcal{H}$-polytope approx.

Figure 2.10: Approximation error introduced by converting an ellipsoid into a box, a zonotope and a polytope.

CHAPTER 3

OPTIMIZATIONS FOR NON-AUTONOMOUS SYSTEMS USING ELLIPSOIDS

In this chapter we will show how to split the computation of flowpipes for non-autonomous systems into an autonomous and a non-autonomous part, such that we obtain chains of linear transformations and chains of Minkowski sums instead of the alternating way described in Chapter 2.2. Afterwards, we will give an efficient way of computing the non-autonomous part by usage of ellipsoids. Lastly, we will propose different ways of reducing support functions in order to reduce the growth of their complexity during the reachability analysis.

First, we will deal with separating the autonomous part and the non-autonomous part of a hybrid system enabling us to use further optimizations on the separated sets [GGM06]. Those optimizations would not be applicable in the original approach due to the alternation between the operation. Further, the sets obtained by the separation are smaller than the original set and thus this approach will also minimize the wrapping effect. The system separation will be done by separately tracking the development of the autonomous part, the non-autonomous part and the necessary bloating in each step.

In order to do this let Ω_0 be the initial set, computed as explained in Chapter 2.2.1. Further, let Ω_i be the set after i steps, $\mathcal{V} = \mathcal{B}_\beta$ be the set needed to cover the external influence in each step and Φ be the matrix representing the linear transformation $e^{A\delta}$. With the reachability algorithm for non-autonomous systems from Chapter 2.2.1 we would compute Ω_i as

$$\Omega_i = \Phi \cdot \Omega_{i-1} \oplus \mathcal{V}.$$

In order to separate the external influence $\mathcal{V}$ from the autonomous system we introduce three new sequences of sets, $\mathcal{A}_i$, $\mathcal{V}_i$ and $\mathcal{S}_i$, with

$$\begin{aligned} \mathcal{A}_0 &= \Omega_0, & \mathcal{A}_{i+1} &= \Phi \cdot \mathcal{A}_i \\ \mathcal{V}_0 &= \mathcal{V}, & \mathcal{V}_{i+1} &= \Phi \cdot \mathcal{V}_i \\ S_0 &= \{0\}, & S_{i+1} &= S_i \oplus \mathcal{V}_i \end{aligned}$$

where $\mathcal{A}_i$ is used to track the autonomous evolution of the system, $\mathcal{V}_i$ represents the growth of the external influence and $\mathcal{S}_i$ tracks the accumulated set of all external inputs and their evolution in the last i steps.

With these sets we can now compute Ω_{i+1} by expanding the autonomous system by the total external influence as

$$\Omega_{i+1} = \mathcal{A}_i \oplus S_i = \Phi^i \Omega_0 \oplus \bigoplus_{j=0}^{i-1} \Phi^j \mathcal{V}.$$

By usage of the distributive law [GGM06] it holds

$$\Phi \cdot (\Omega \oplus \mathcal{V}) = (\Phi \cdot \Omega) \oplus (\Phi \cdot \mathcal{V}).$$

The correctness of this computation follows directly from the properties of linear transformation and Minkowski sum.
By applying this distribution of linear transformations over Minkowski sums the correctness of our new way of computing Ω_{i+1} is easy to show. It holds

$$\Omega_i = \Phi \cdot \Omega_{i-1} \oplus \mathcal{V} = \Phi^i \Omega_0 \oplus \bigoplus_{j=0}^{i-1} \Phi^j \mathcal{V}.$$

The corresponding modification of the flowpipe-computation can be found in Algorithm 2.

Algorithm 2 Reachability of discrete linear time-invariant systems

Input: Lin. trans. Φ, initial sets Ω_0, $\mathcal{V}$, an integer N
Output: The first N segments of this sequence.

1: $\mathcal{A}_0 \leftarrow \Omega_0$
2: $\mathcal{V}_0 \leftarrow \mathcal{V}$
3: $\mathcal{S}_0 \leftarrow \{0\}$
4: **for** i from 1 to $N-1$ **do**
5: $\quad \mathcal{A}_i \leftarrow \Phi \mathcal{A}_{i-1}$
6: $\quad \mathcal{S}_i \leftarrow \mathcal{S}_{i-1} \oplus \mathcal{V}_{i-1}$
7: $\quad \mathcal{V}_i \leftarrow \Phi \mathcal{V}_{i-1}$
8: $\quad \Omega_i \leftarrow \mathcal{A}_i \oplus \mathcal{S}_i$
9: **end for**
10: **return** $\{\Omega_0, \ldots, \Omega_{N-1}\}$

We now have an alternative way of computing Ω_{i+1} with a separated computation of the autonomous and non-autonomous part and as explained before we do not have an alternation between linear transformations and Minkowski sums. Furthermore, as approximations are mostly used for Minkowski sums, we strongly reduce the wrapping effect as the impact of wrapping strongly depends on the size of the objects and in most cases the external input $\mathcal{V}$ is rather small compared to the autonomous system. This way of computing Ω_{i+1} has been shown to improve the computation of reachable sets of linear time-invariant systems both theoretically and empirically [GGM06].

Note that this decomposition is only possible if the system is time-invariant. If the system is not time-invariant there is currently no known way to apply such a decomposition [GGM06].

The autonomous part $\mathcal{A}_i$ can now be efficiently computed as a chain of matrix multiplications but the non-autonomous part $\mathcal{S}_i$ still needs to handle a lot of Minkowski sums and linear transformations. For this reason we will show a way of how to reduce the computational effort needed to compute $\mathcal{S}_i$ by usage of ellipsoids in the next chapter.

3.1 Computation of the Non-Autonomous Part

Earlier in Chapter 2.3 we explained that the efficiency of computing the Minkowski sum of two or more sets strongly depends on the used representation. For support functions representing a ball it is a simple addition of their radii and for polytopes we would need to respect all combinations of points from the different polytopes. Even though it sounds reasonable here to use support functions, there is a huge disadvantage. The Minkowski sum itself is easy to compute, but we still need to evaluate both branches used in the Minkowski sum. Looking at the way $\mathcal{S}_i$ is computed we see that this will involve long chains of linear transformations that need to be evaluated. This may yield high costs for evaluating the support function that might not not visible at first. This problem have a greater impact when we need to compute flowpipes with many segments and thus obtain large support function trees.

Looking only at the form of the external input, we see that using a support function to represent $\mathcal{V}$ would be the easiest way to compute the Minkowski sum of all external inputs. Unfortunately, as explained above, the computation will slow down after some steps. Polytopes are not suited either due to their high complexity on Minkowski sums. That is why we will use ellipsoids to represent $\mathcal{V}_i$ as well as S_i. For ellipsoids we can efficiently compute the current sets while avoiding high evaluation costs. During the iteratively computing the flowpipe we always have a closed representation of the ellipsoid representing the external input $\mathcal{V}_i$ as well as the sum of all external inputs $\mathcal{S}_i$. Thus, using ellipsoids yields in a constant evaluation time in all iteration steps for the external input while minimizing the costs to obtain S_i.

Assuming that the external input $\mathcal{V} = \mathcal{B}_\beta$ is a ball we are already given our initial ellipsoid, as balls are a special case of ellipsoid. From the previous explanations on ellipsoids in Chapter 2.3.3 we already know that the linear transformation of an ellipsoid is an ellipsoid as well and therefore the computation of $\mathcal{S}_i$ boils down to the Minkowski sum of $i-1$ ellipsoids which, as shown in Chapter 2.3.3, can be efficiently computed and tightly approximated in direction l by

$$\bigoplus_{j=0}^{i-1} \Phi^j \mathcal{V} \subseteq \left(\sum_{j=0}^{i-1} \langle l, \Phi^j Q_j {\Phi^j}^T l \rangle^{1/2} \right) \cdot \sum_{j=0}^{i-1} \left(\frac{1}{\langle l, \Phi^j Q_j {\Phi^j}^T l \rangle^{1/2}} \Phi^j Q_j {\Phi^j}^T \right)$$

Another advantage of this way of computing the non-autonomous part is that we will get a closed form representation that can be evaluated in an arbitrary direction by a single matrix-vector multiplication. Because of that, the approach works well if

the autonomous part is represented by a support function. Computing Ω_i can then be realized by converting S_i into a support function which works in constant time and then building the Minkowski sum of those two support functions. Evaluating Ω_i has now the costs of evaluating an autonomous system plus a single matrix-vector multiplication. However, even if the representation of the autonomous part is not a support function it is easy to convert S_i into a given representation fitted for the Minkowski sum with the autonomous part as shown in Chapter 2.3.3.

One thing to notice is that this approach will not work efficiently on systems where each segment is only evaluated a small number of times in comparison to its dimension. In this cases the costs for construction of the ellipsoid are higher than the costs of evaluating a support function representing the same set. Thus, in this cases we would advise to use support functions to compute the non-autonomous part instead of ellipsoids. However, those kind of systems are rare in practice. A more detailed explanation on the number of evaluations that may occur on a segment or during the flowpipe construction can be found in Chapter 3.2.2.

Example 3.1
As an example we will compare the number of necessary matrix-vector multiplications for evaluating the non-autonomous part of a flowpipe analysis with ellipsoids and support functions. As these multiplications have a huge influence on the runtime of the flowpipe construction they are a feasible value for comparisons.
Let us consider a flowpipe with 101 segments which each describes a 3-dimensional state set. Further, each segment is evaluated four times, twice for the invariance of the location and twice for the guard of an outgoing transition. Evaluating the non-autonomous part of the 101-th segment once with support functions we would need to compute $\sum_{i=1}^{100} i = 5{,}050$ matrix-vector multiplications to cover all linear transformations in the support function tree of $\mathcal{S}_{101} = \bigoplus_{j=0}^{100} \Phi^i \mathcal{V}$. For ellipsoids this would be a single matrix-vector multiplication. However, computing the ellipsoid of the 100-th segment would need $100 \cdot (3 \cdot 3) = 900$ matrix-vector multiplications to cover the linear transformations and approximately 200 matrix-vector multiplications to cover the necessary Minkowski sums (very coarse estimation). In conclusion, for the evaluation of the non-autonomous part with usage of ellipsoids we would need at most 1,100 matrix-vector multiplications, whereas support functions would need 5,050. However, for ellipsoids the actual costs are even lower as the ellipsoid of the 100-th segment would already be known in an iterative implementation. Thus, the necessary number of matrix-vector multiplications would reduce to less than 15.

3.1.1 Exact Arithmetic

Exact arithmetic are an interesting field of research on verification topics. One of the major advantages of exact arithmetic is the non-existence of numerical problems. Numbers are represented by a fraction and can be arbitrary precise. Thus, we are not bound to the precision of common data-types. The drawback of this number representation is that numbers can be arbitrary large in the sense of memory and therefore they can have a major impact on computational time.

Even though the computation using Minkowski sums on ellipsoids is theoretically fast there is a major drawback. Due to the way of computing the Minkowski sum of el-

lipsoids the length of the numbers in the shape matrix blows up enormously when using exact arithmetic leading to a massive slowdown of the computation. Note that using floating point arithmetics with a fixed bit-length, longer numbers do not need more memory to be stored but more time to be processed. However, this blow-up in size can be avoided by over-approximating the slow-computing ellipsoid by a slightly larger one, where the numbers need less time to be processed.

A common technique to decrease computational time in the presence of exact arithmetic is rounding. However, one has to ensure that the results are still correct after rounding. Which brings us to the main part of this section. We will be rounding the shape-matrix of an ellipsoid such that it is an over-approximation of the original ellipsoid. The precision of the following approximation depends on the digit where the numbers will be rounded. Further, the over-approximating ellipsoid will be slightly grown and rotated.

To ensure that the new ellipsoid $\mathcal{E}(q,Q')$ is an over-approximation of the original ellipsoid $\mathcal{E}(q,Q)$ we have to show that the following holds

$$\begin{aligned}\rho(l \mid \mathcal{E}(q,Q)) = \langle l,q\rangle + \sqrt{\langle l,Ql\rangle} \leq \langle l,q\rangle + \sqrt{\langle l,Q'l\rangle} = \rho(l \mid \mathcal{E}(q,Q')) \\ \Leftrightarrow \langle l,Ql\rangle \leq \langle l,Q'l\rangle,\end{aligned}$$

for all directions $l \in \mathbb{R}^d$. Without loss of generality, for the following equations we assume that all entries c_j,k are positive (otherwise round up instead of down on that specific entry). Further, we assume l to be normalized.

$$\begin{aligned}\langle l, Ql\rangle &= \left\langle \begin{pmatrix} l_1 \\ \vdots \\ \vdots \\ l_n \end{pmatrix}, \begin{pmatrix} a_1 & c_{1,2} & \cdots & c_{1,n} \\ c_{1,2} & a_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & c_{n-1,n} \\ c_{1,n} & \cdots & c_{n-1,n} & a_n \end{pmatrix} \begin{pmatrix} l_1 \\ \vdots \\ \vdots \\ l_n \end{pmatrix} \right\rangle \\ &= \sum_{i=1}^{n} \left(a_i \cdot l_i^2\right) + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \left(2 \cdot c_{j,k} \cdot l_j \cdot l_k\right) \\ &= \sum_{i=1}^{n} \left(a_i \cdot l_i^2\right) + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \left(2 \cdot \lfloor c_{j,k} \rfloor \cdot l_j \cdot l_k + 2 \cdot (c_{j,k} - \lfloor c_{j,k} \rfloor) \cdot l_j \cdot l_k\right) \\ &\quad l \text{ is normalized, thus it holds} \sum_{i=1}^{m} l_i^2 = 1 \Rightarrow l_j \cdot l_k \leq 0.5,\ l_j \neq l_k \\ &\leq \sum_{i=1}^{n} (a_i \cdot l_i^2) + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \left(2 \cdot \lfloor c_{j,k} \rfloor \cdot l_j \cdot l_k + 2 \cdot (c_{j,k} - \lfloor c_{j,k} \rfloor)\right) \\ &= \sum_{i=1}^{n} (a_i \cdot l_i^2) + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \left(2 \cdot \lfloor c_{j,k} \rfloor \cdot l_j \cdot l_k + (c_{j,k} - \lfloor c_{j,k} \rfloor) \cdot \underbrace{(l_1^2 + \ldots + l_n^2)}_{=1}\right) \\ &= \sum_{i=1}^{n} (a_i \cdot l_i^2) + \sum_{j=1}^{n-1} \sum_{k=j+1}^{n} \left(2 \cdot \lfloor c_{j,k} \rfloor \cdot l_j \cdot l_k\right)\end{aligned}$$

$$
\begin{aligned}
&\quad + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}((c_{j,k} - \lfloor c_{j,k}\rfloor)\cdot(l_1^2 + \ldots + l_n^2)) \\
&= \sum_{i=1}^{n}(a_i \cdot l_i^2) + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(2\cdot\lfloor c_{j,k}\rfloor \cdot l_j \cdot l_k) + \sum_{m=1}^{n}\sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(l_m^2\cdot(c_{j,k} - \lfloor c_{j,k}\rfloor)) \\
&= \sum_{i=1}^{n}\left(\left(a_i + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(2\cdot(c_{j,k} - \lfloor c_{j,k}\rfloor)\right)\cdot l_i^2\right) + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(2\cdot\lfloor c_{j,k}\rfloor \cdot l_j \cdot l_k) \\
&\leq \sum_{i=1}^{n}\left(\underbrace{\left\lceil a_i + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(2\cdot(c_{j,k} - \lfloor c_{j,k}\rfloor)\right\rceil}_{a_i'}\cdot l_i^2\right) + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(2\cdot\underbrace{\lfloor c_{j,k}\rfloor}_{c_{j,k}'}\cdot l_j \cdot l_k) \\
&\quad \text{introduce new variables } a_i', c_{j,k}' \\
&= \sum_{i=1}^{n}(a_i' \cdot l_i^2) + \sum_{j=1}^{n-1}\sum_{k=j+1}^{n}(2\cdot c_{j,k}' \cdot l_j \cdot l_k) \\
&= \left\langle \begin{pmatrix} l_1 \\ \vdots \\ \vdots \\ l_n \end{pmatrix}, \begin{pmatrix} a_1' & c_{1,2}' & \cdots & c_{1,n}' \\ c_{1,2}' & a_2' & \ddots & \vdots \\ \vdots & \ddots & \ddots & c_{n-1,n}' \\ c_{1,n}' & \cdots & c_{n-1,n}' & a_n' \end{pmatrix}\begin{pmatrix} l_1 \\ \vdots \\ \vdots \\ l_n \end{pmatrix}\right\rangle \\
&= \langle l, Q'l\rangle
\end{aligned}
$$

This approximation can be efficiently computed as all used operations behave well, even for rational arithmetic. Furthermore, we can directly influence the accuracy of the computation by specifying the digit where the numbers are rounded.

In addition to getting rid of 'slow' numbers, we can try to slow down the increase of complexity of numbers used. The main course of increasing complexity of numbers during the computation of Minkowski sums of ellipsoids are the involved multiplications with the results of a square-root. This can be avoided by limiting the influence of the square-root in the first place. One possible way to do this is to round the numbers before multiplication resulting in the following computation of Minkowski sums. As a reminder, normally we compute the Minkowski sum of ellipsoids as

$$Q_l^+ = \left(\sum_{i=1}^{k}\langle l,Q_i l\rangle^{\frac{1}{2}}\right)\left(\sum_{i=1}^{k}\frac{1}{\langle l,Q_i l\rangle^{\frac{1}{2}}}Q_i\right).$$

With exact arithmetic one might want to use the following over-approximation

$$Q_l^+ = \left(\sum_{i=1}^{k}\lceil\langle l,Q_i l\rangle^{\frac{1}{2}}\rceil\right)\left(\sum_{i=1}^{k}\frac{1}{\lfloor\langle l,Q_i l\rangle^{\frac{1}{2}}\rfloor}Q_i\right),$$

with $\langle l,Q_i l\rangle > 0$. The correctness follows directly from $\lceil\langle l,Q_i l\rangle^{\frac{1}{2}}\rceil \geq \langle l,Q_i l\rangle^{\frac{1}{2}}$ and $\frac{1}{\lfloor\langle l,Q_i l\rangle^{\frac{1}{2}}\rfloor} \geq \frac{1}{\langle l,Q_i l\rangle^{\frac{1}{2}}}$.

From our implementation and tests we have concluded that those approximations have a huge impact on the computational time while the growth of the sets is is negligible small for a sufficiently small digit set as rounding point.

3.2 Optimizations for Reachability with Support Functions

In addition to using ellipsoids as optimizations for the computation of the non-autonomous part, we propose some optimizations for the autonomous part in the presence of support functions. As stated in Chapter 3 support functions and ellipsoids work well together. However, the computation of $\mathcal{A}_i$ using ellipsoids still has the drawbacks of large support function trees. In order to reduce the complexity of the involved support functions we will propose three ways to reduce the size of the trees and thus speed up the reachability computation.

3.2.1 Reduction of Linear Transformation Chains

As we explained in the previous Chapter, we can compute the reachable set $\mathcal{A}_i$ by

$$\mathcal{A}_i = \Phi\mathcal{A}_{i-1} = \Phi^i\Omega_0,$$

which yields long chains of linear transformations. Thus, evaluating $\Phi^i\Omega_0$ with support functions needs i matrix-vector multiplications. It is easy to show that this is not efficient if we need to evaluate $\mathcal{A}_i$ or the corresponding Ω_i multiple times. In order to reduce the number of necessary multiplications, thus efficiently reducing the support function tree, we introduce a simple reduction method that will be referred to as *linear transformation reduction*. We will search the support function tree for chains of successive linear transformations and replace those chains by a single linear transformation. Starting from the leaves, we will search the tree for chains of i successive linear transformations Φ. This chain will then be replaced by the linear transformation $\Phi' = \Phi^i$. To reduce the costs of computing Φ' we suggest to choose $i = 2^n$, $n \in \mathbb{N}$. For those i we can efficiently compute Φ^i with only n matrix multiplications.

In an optimal scenario this will reduce the size of the support function tree T to $\mathcal{O}(\log_n size(T))$. However, the effect on the reachability analysis depends strongly on the ratio between evaluations and the number of variables of the system. For a high number of variables and low number of evaluations this reduction can even have a negative impact on the computation. The reason for this is that the reduction might be more expensive than computing the flowpipe without the reduction. Furthermore, the reduced tree has in general a higher storage complexity due to the overhead introduced by the newly generated matrices.

Example 3.2
Consider a support function consisting of a polytope which is 4 times linear transformed with the same transformation matrix Φ. The corresponding support function tree is shown in Figure 3.1.
Assume we want to reduce the support function if two successive linear transformation use the same matrix. In order to do this, we traverse the support function tree

(1) starting from the bottom. In this scenario we would first obtain support function (2) by reducing the second and the fourth linear transformation in the support function and replace them by Φ^2. In the reduced support function (2) both linear transformations again use the same matrix Φ^2, therefore we could again use the linear transformation reduction and obtain support function (3).

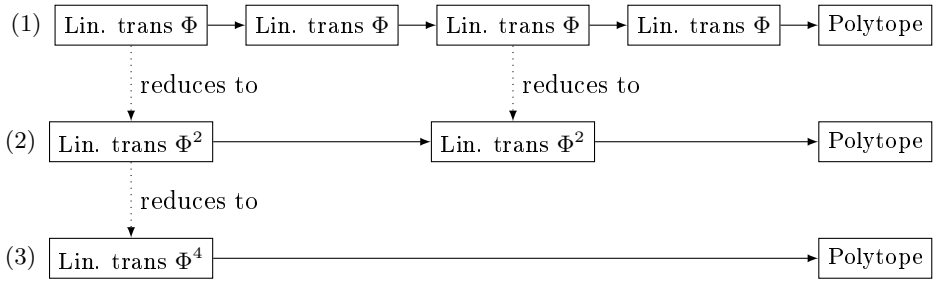


Figure 3.1: Illustration of the linear transformation reduction performed on a support function tree.

3.2.2 Reduction of Jump Complexity

In Chapter 2.2 we already explained how to compute the jump successors of a flowpipe. We compute the union of all sets satisfying the jump guards, and then apply the reset function corresponding to the jump to this aggregated set. For most representations this not a complex problem. For support functions however, the union of those segments leads to a major increase in computation complexity. Due to the way support functions are implemented, the support function tree after the jump contains the trees of all support functions that satisfied the jump guard. Thus, evaluating the first segment after the jump is more complex than evaluating all segments involved in the jump at once. In the following we are going to reduce this significant growth by reducing the support function before and after the jump.

Post-Jump Reduction

A simple approach to get rid of this aggregation problem is to simply over-approximate the set after the jump was taken by a $\mathcal{H}$-polytope. This way we remove the whole support function tree, leaving us with only a polytope-representing support function. We will refer to this approach as *post-jump reduction*.

In order to approximate the current set Ω by a polytope we need to fix the number of halfspaces that will be used as this directly influences the costs and precision of the approximation. A common approach is to evaluate the support function ρ_Ω in n evenly distributed directions for each pair of dimensions and build the $\mathcal{H}$-polytope from the induced halfspaces.
The construction of such a $\mathcal{H}$-polytope would need $n \cdot \binom{d}{2}$ evaluations. Due to the possibly high number of necessary evaluations this approximation might be expensive to compute. Thus, applying the post-jump reduction and then compute the flowpipe

might be more expensive than computing the whole upcoming flowpipe with the original support function. Thus, in order to know where to efficiently reduce the support function a metric is needed.

In order to make a founded assumption about whether it is useful to reduce the current support function we need knowledge the reduction costs as well as the expected costs for the flowpipe construction.
First, we need to make a coarse assumption of the number of expected evaluations for upcoming flowpipe for the current location. Note that only the next flowpipe is of interest here and not all upcoming flowpipes. The maximal number of upcoming segments can be computed by $\frac{N}{\delta}$ where N is the time horizon and δ is the step size. In addition, we need to know how many evaluations will be computed per segment. For each segment we need to check its containment in the invariant as well as test if the guards of outgoing transitions are satisfied. This results in one evaluation for each invariant as well as one evaluation for each outgoing guarded transition. With these values, we can calculate an estimation of the number of evaluations necessary to compute the upcoming flowpipe by

$$\#Eval = \frac{N}{\delta} \cdot (\#Guards + \#Invariants)$$

Now that the number of upcoming evaluations can be estimated, we only need to know the costs c of evaluating the current support function. To make a reasonable assumption for these costs, we will use the number of matrix-vector multiplications needed for an evaluation to measure the costs.
Commonly the initial set X_0 is specified by a polytope. Thus, the support function representing Ω_0 has an underlying polytope and, as evaluating this polytope involves solving a linear optimization problem, we need to respect this fact as well in our cost estimation. Further, all operations except for linear transformations are computable in constant time and only occur sparsely compared to linear transformations. Therefore, these operations will be omitted from the cost estimation. This gives us a rather fast way to analyze the tree introduced by the support function in order to obtain the estimated costs. We just have to sum the cost for all linear transformations, which all have costs of 1 and the costs for all polytope evaluations which have a cost of $n \cdot \max(m,n)^2$. Note that $\max(m,n)^2$ refers to an estimation of the average number of matrix multiplications needed by the simplex algorithm where the problem is given by a $m \times n$ matrix [ST01]. If the system is non-autonomous we also need to respect ellipsoids which can be evaluated with a single linear transformation as well. A recursive algorithm computing the costs c can be found in Algorithm 3.

Now that we are able to estimate the cost per evaluation as well as the number of upcoming evaluations we can give a closed formula to decide if a post-jump reduction would be efficient. We will use the reduction by over-approximating with a $\mathcal{H}$-polytope only if the following in-equation is satisfied

$$\underbrace{\#Eval \cdot c}_{\substack{\text{evaluationcosts} \\ \text{without reduction}}} \geq \underbrace{n \cdot \binom{d}{2} \cdot c}_{\text{costs of reduction}} + \underbrace{d \cdot (n \cdot \binom{d}{2})^2 \cdot \#Eval}_{\text{evaluationcosts with reduction}},$$

where $d \cdot (n \cdot \binom{d}{2})^2$ is the evaluation cost of the support function corresponding to the newly generated polytope.

Algorithm 3 Evaluation cost estimation

Input: Support function tree
Output: Number of matrix-vector multiplications needed for an evaluation a support function.

```
1: c ← 0
2: if nodeIsLinTrans ∨ nodeIsEllipsoid then
3:     c ← 1
4: end if
5: if nodeIsPolytope then
6:     c ← n · max(m,n)^2
7: end if
8: for successors of node do
9:     c ← c + estimateCosts(successor)
10: end for
11: return {Ω_0, ..., Ω_{N-1}}
```

Note that this is a rather coarse estimation of the actual costs. For a more precise calculation one would need to respect the evaluation costs of all operations as well as get a better estimation on the evaluation costs of the maximization problem. In addition the metric depends on the time horizon N. We later on propose a way to dynamically compute an individual time horizon for each location in order to improve the effects of our metric. The estimation of obtained hyperplanes $n \cdot \binom{d}{2}$ can be reduced by avoiding an evaluating the same direction multiple times.

Example 3.3
For $n = 8$ the number of hyperplanes can be reduced from $8 \cdot \binom{d}{2}$ to $4 \cdot \binom{d}{2} + 2 \cdot d$ if the evaluation directions are evenly distributed and we evaluate in direction of the involved axis.

Pre-Jump Reduction

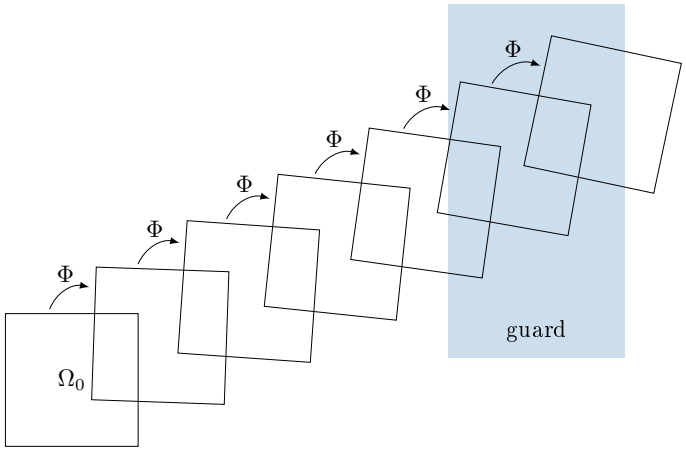
One problem of the post-jump reduction is its the bad scalability with higher number of dimensions and fewer numbers of guards and invariants. Thus we propose an additional approach aimed to reduce the support functions complexity before the jump. This approach will be referred to as the *pre-jump reduction*.

We already explained that the complexity of all segments satisfying a guard will be combined in the first segment after the jump. Thus, the number of operations necessary to evaluate this segment can be rather large depending on the complexity of the segments satisfying the guard. Due to this relation, we aim on reducing the complexity of guard-satisfying segments as follows.
Once an outgoing transition is discovered to be enabled during the flowpipe construction the current segment Ω_i which was computed as $\Omega_i = \Phi\Omega_{i-1}$ will be replaced by a direct successor of the initial segment Ω_0

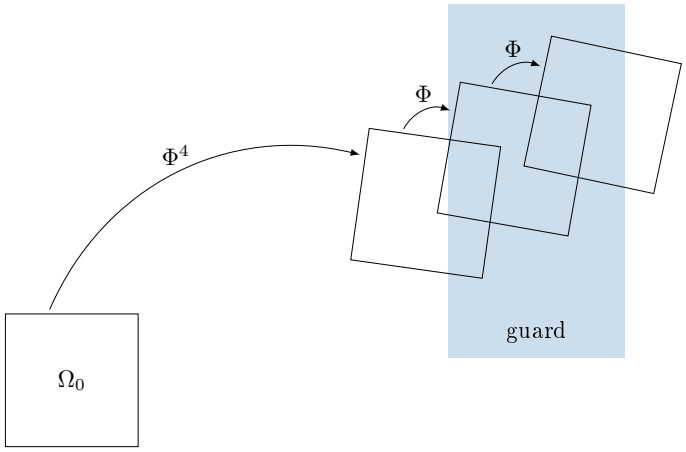
$$\Omega'_i = \Phi'\Omega_0,$$

where $\Phi' = \Phi^i$. The following segments are then computed based on Ω'_i. This way we can reduce the complexity of all segments involved in the jump. However by directly

computing Ω_i' from Ω_0 we neglect all previous intersections with invariants and thus, we might obtain an over-approximation of the original segment Ω_i. An illustration of this idea is shown in Figure 3.2.



(a) original



(b) reduced

Figure 3.2: Reducing complexity of a support function before a jump

CHAPTER 4
EXPERIMENTAL RESULTS

We will now analyze the influence of our approaches on the reachability analysis of the HyPro tool. First, we give a detailed description of the used benchmarks as well as used parameters and afterwards discuss the results we got on those benchmarks with respect to the optimizations from Chapter 3.

4.1 Benchmarks

In this section we will give a detailed description of the hybrid systems, as well as the corresponding hybrid automata and the reachability settings that were used for our tests. For some benchmarks the corresponding hybrid automata and matrices were moved to Appendix B due to their size.

4.1.1 Bouncing Ball

The classical bouncing ball benchmark[1], which was already used in Chapter 2.1 models a ball that is dropped from a predefined height. After a certain time of falling it hits the ground and bounces back off into the air and starts to fall again. While hitting the ground the ball will lose part of its energy and thus looses a bit of momentum. This physical phenomenon can be represented by a hybrid automaton as seen in Figure 4.1

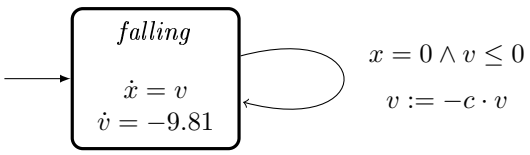


Figure 4.1: Hybrid automaton of the bouncing ball benchmark

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

Settings

We consider the initial set

$$x \in [10; 10.2]$$
$$v = 0.$$

Further, we used a time horizon of $N = 3$, step-size $\delta = 0.01$ and a jump depth of 3. The constant c is set to $c = 0.75$ and the set of bad states is the set of all states where $v \geq 10.7$.

4.1.2 Two Tanks

The two tanks benchmark describes the controlling of two water-tanks, where the input and output can be regulated via certain valves. The first tank is being filled from two different sources: a constant flow source and a second source equipped with a controlled value $valve_1$, with flows Q_0 and Q_1 respectively. At the bottom of tank 1 an attached drain allows the liquid to flow directly into tank 2 with flow Q_A. Tank 2 has two drains attached. One connected to a pump to assure a constant liquid outflow Q_B and a second one which's outflow Q_2 is controlled by an electronic valve $valve_2$. $Valve_1$ and $Valve_2$ can both take states On/Off. This results in four possible discrete modes for the hybrid automaton as shown in Figure 4.2 [His01].

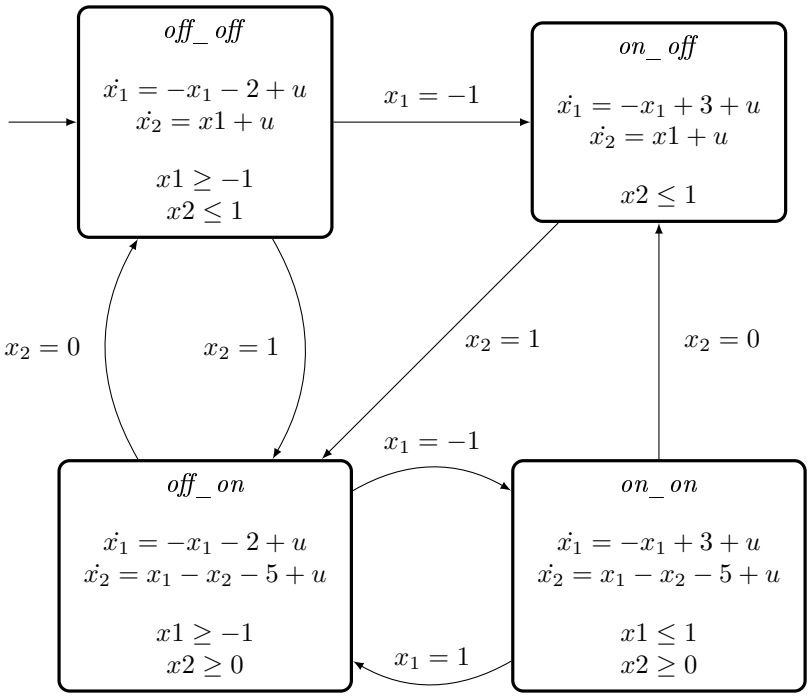


Figure 4.2: Hybrid automaton of the two tanks benchmark

The liquid levels in tank i are given by x_i and the dynamics of the system is defined

by the following differential equations:

$$\dot{x}_1 = \begin{cases} -x_1 - 2 + [-0.1,0.1] & \text{if } valve_1 \text{ is } Off \\ -x_1 + 3 + [-0.1,0.1] & \text{if } valve_1 \text{ is } On \end{cases}$$
$$\dot{x}_2 = \begin{cases} x_1 + [-0.1,0.1] & \text{if } valve_2 \text{ is } Off \\ x_1 - x_2 - 5 + [-0.1,0.1] & \text{if } valve_2 \text{ is } On \end{cases}$$

Settings

We consider the initial set

$$x_1 \in [1.5,2.5]$$
$$x_2 = 1,$$

with initial state off_off. Further, we used a time horizon of $N = 2$, step-size $\delta = 0.01$ and a jump depth of 2. The set of bad states are all states, where $x_2 <= -0.7$.

4.1.3 Rod Reactor

The rod reactor example[1] describes a crucial system inside a nuclear reactor. Inside a reactor tank there are two independent nuclear control rods. The goal is to control the coolant temperature x in the tank by putting in or taking out the rods and avoid reaching the critical temperature. The rods rod_1 and rod_2 both have their individual cooling dynamics and are both attached to a clock c_1 and c_2 respectively which measures the time elapsed since the rod was last used. The hybrid automaton corresponding to the benchmark is shown in Figure 4.3

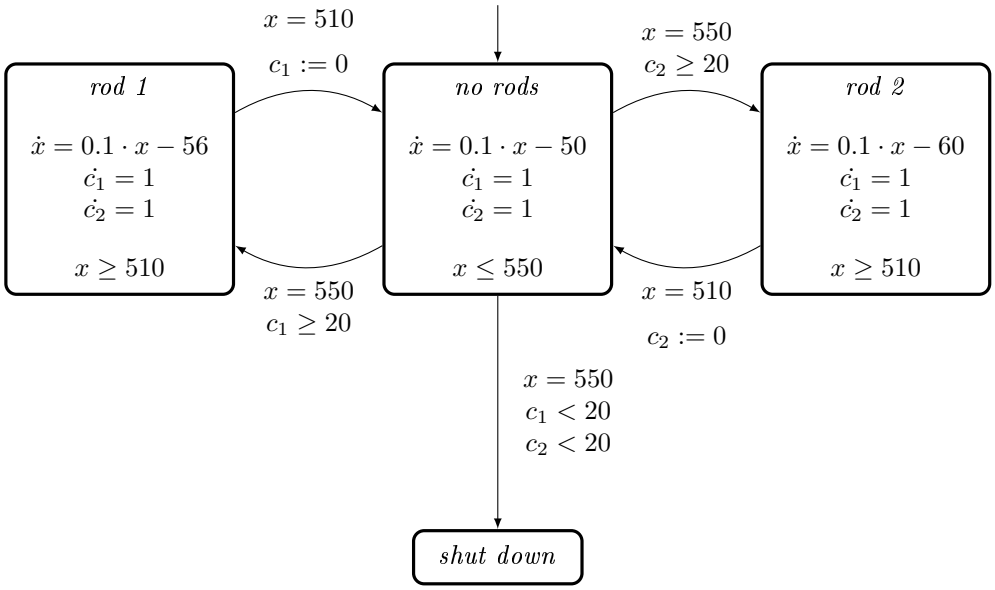


Figure 4.3: Hybrid automaton of the rod reactor benchmark

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

Settings

We consider the initial set

$$x = 510$$
$$c1,c2 = 20,$$

with initial location *no_rods*. Further, we used a time horizon of $N = 50$, step-size $\delta = 0.1$ and a jump depth of 4. The set of bad states are all states in location "shut down".

4.1.4 Cruise Control

The cruise controller benchmark models a controller that is build into a vehicle. Its goal is to keep the vehicles velocity at a certain level. The controller can access an *accelerator* and *service brake* and an *emergency brake*. Both brakes can be activated simultaneously. Further, if the car is neither actively accelerating nor braking the vehicle will slowly decelerate due to the braking behavior of the engine itself. The hybrid automaton describing this scenario with fixed acceleration and deceleration values can be found in Chapter B.3 [Oeh11].

Settings

We consider the initial set

$$v \in [15,40]$$
$$t \in [0, 2.5]$$
$$x = 0,$$

with initial location loc_1. Further, we used a time horizon of $N = 100$, step-size $\delta = 0.5$ and a jump depth of 5. The set of bad states are the set of states where $v \leq -2$.

4.1.5 5-Dimensional Linear Switching System

The 5-dimensional linear switching system benchmark[1] models a purely fictive system with randomly generated flows. It is a piecewise linear system with different controlled continuous dynamics. The matrices A_i,B_i, with $i \in \{1,\ldots,5\}$ are generated randomly and the generated system is then stabilized with a LQR controller to ensure a convergence to a stable attracting region. The transitions are determined heuristically by means of simulations. The proposed benchmark consists of 5 locations and 5 transitions ordered as a cycle. The continuous dynamics in each mode q_i, $i \in \{1,\ldots 5\}$ is described by the ODE

$$\dot{x} = A_i x + B_i u,$$

where $x \in \mathbb{R}^5$ is the state vector, u is an input signal confined in compact bounded set U and the matrices $A_1,\ldots,A_5$ and $B_1,\ldots,B_5$ are defined as in Chapter B.1. The hybrid automaton representing the 5-dimensional switching system is given in Figure 4.4

[1] `https://ths.rwth-aachen.de/research/projects/hypro/5-dimensional-switching-linear-system/`

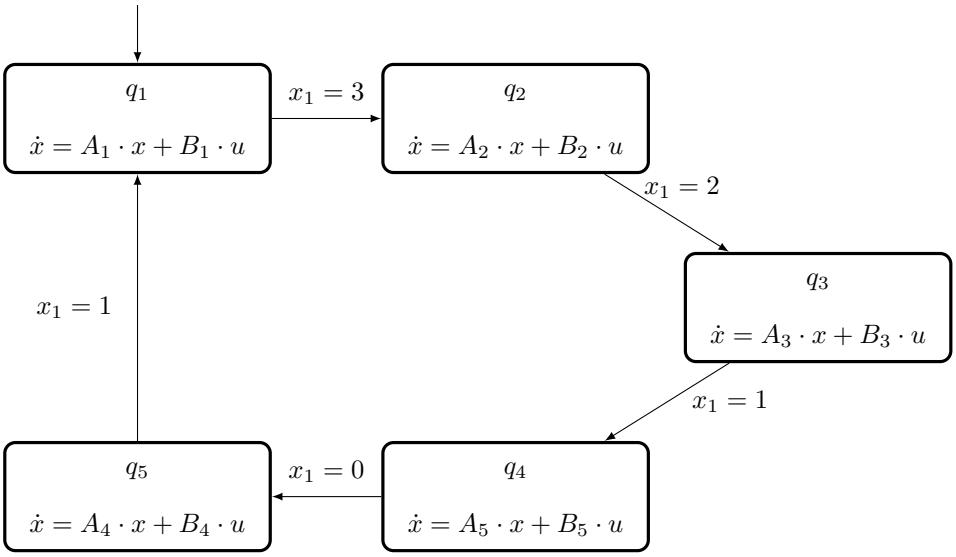


Figure 4.4: Hybrid automaton of the 5-dimensional switching system

Settings

We consider the initial set

$$\begin{aligned} x_1 &= 3.1 \\ x_2 &= 4 \\ x_3 &= 0 \\ x_4 &= 0 \\ x_5 &= 0 \\ u &\in [-1,1], \end{aligned}$$

with initial location q_1. Further, we used a time horizon of $N = 0.13$, step-size $\delta = 0.003$ and a jump depth of 4.

4.1.6 Three-Vehicle Platoon

The three-vehicle platoon benchmark models a distance control system to avoid collisions within the platoon. We consider a platoon of 3 carrier vehicles guided by an additional leader vehicle. All vehicles are able to communicate with each other and they are trying to keep the spacing between them at a fixed distance. Due to heavy rain, the communication may be disrupted. The spacing error e_i is defined as the difference between the distance d_i of the truck i to its predecessor and a reference distance $d_{ref,i}$.

$$e_i = d_i - d_{ref,i}.$$

The goal of the reachability analysis is to determine a lower bound for $d_{ref,i}$ that ensures a collision-free driving. The dynamics of the platoon is described by the following ODE

$$\dot{x}(t) = Ax(t) + Ba_L(t),$$

where A is a constant system matrix, B is a constant input matrix and a_L is the acceleration of the leader considered here as an external input. The state vector is $x = [e_1,\dot{e_1},a_1,e_2,\dot{e}_2,a_2,e_3,\dot{e}_3,a_3]$ where a_i is the acceleration of vehicle i. In case of no communication problems A and B are given by A_c and B_c and if the communication is disrupted the matrices describing the describing the dynamics are given by A_n and B_n as defined in Chapter B.2. The hybrid automaton corresponding to the benchmark can be found in Figure 4.5 [MMH+11].

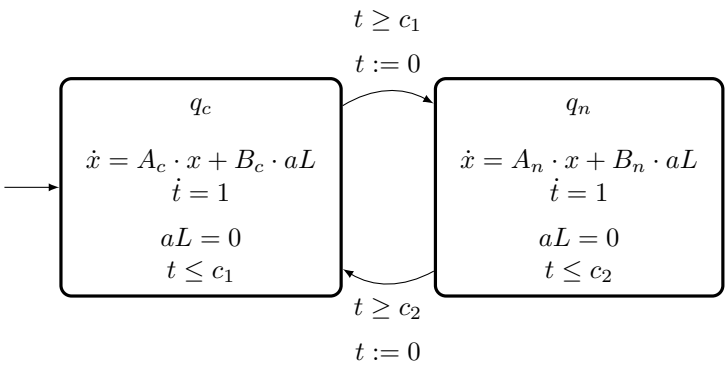


Figure 4.5: Hybrid automaton for the three-vehicle platoon benchmark

Settings

We consider the inital set

$$\begin{aligned} e_i &\in [0.9,1.1],\ i \in \{1,\dots,3\} \\ \dot{e_i} &\in [0.9,1.1],\ i \in \{1,\dots,3\} \\ a_i &\in [0.9,1.1],\ i \in \{1,\dots,3\} \\ c_1,c_2 &= 2 \\ a_L &= 0, \end{aligned}$$

with initial location q_c. Further, we used a time horizon of $N = 12$, step-size $\delta = 0.2$ and a jump depth of 2. The set of bad states are the states where $e_1 \geq 1.7$.

4.1.7 Filtered Oscillator

The filtered oscillator benchmark[1] models a two-dimensional switched oscillator with variables x and y combined with a 4-th order filter with variables x_1, x_2, x_3 and z. The filter smoothens the input signal x and has the output signal z. At the switching planes the system changes its mode but does not modify the variables. The corresponding hybrid automaton with fixed values can be found in Figure 4.6.

Settings

We consider the initial set

$$x \in [0.2,0.3]$$

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

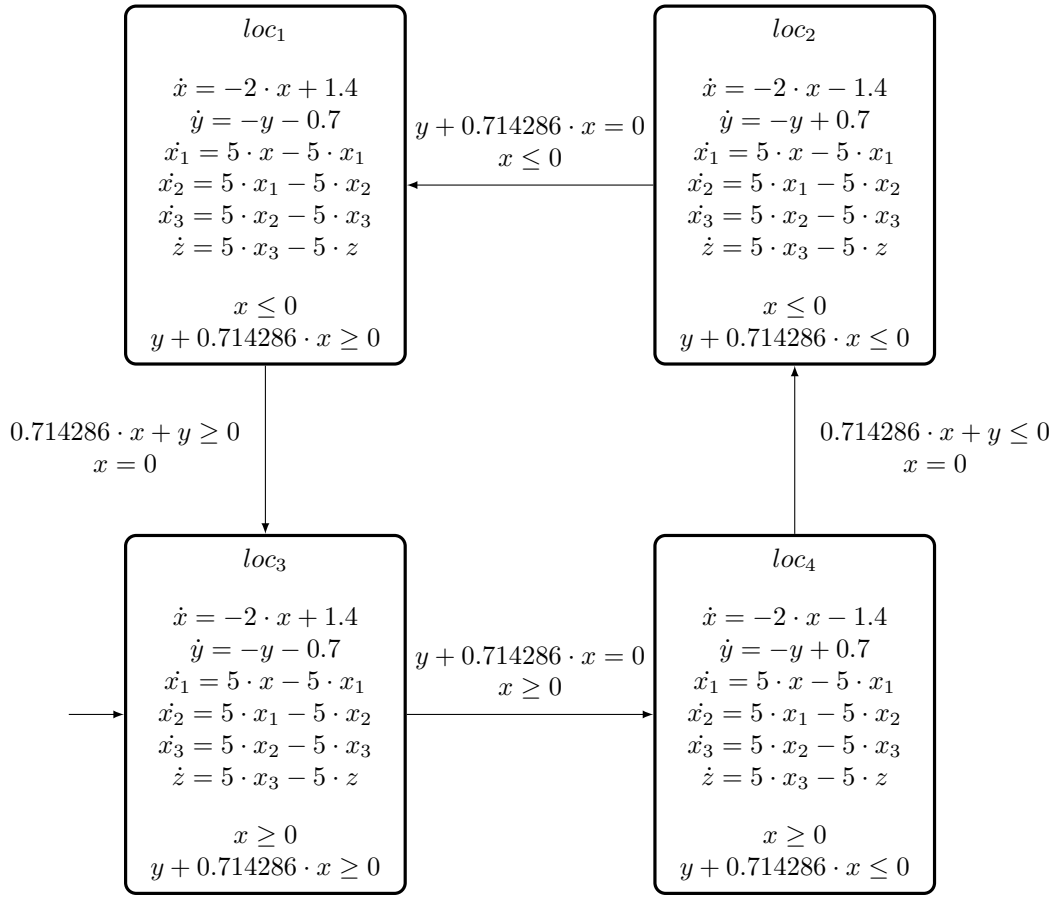


Figure 4.6: Hybrid automaton for the filtered oscillator benchmark

$$y \in [-0.1,0.1]$$
$$z,x_1,x_2,x_3 = 0$$

with initial location loc_3. Further, we used a time horizon of $N = 4$, step-size $\delta = 0.1$ and a jump depth of 5. The set of bad states are the states where $y \geq 0.5$.

4.2 Evaluation

All benchmarks described in Chapter 4.1 were tested within the implementation of HyPro. We used different combinations of our optimization approaches and used boxes and $\mathcal{H}$-polytopes as reference values for the runtime. Unfortunately some optimizations like the influence of ellipsoids on non-autonomous systems and the variance of the linear transformation reduction parameter could not be tested on all benchmarks due to high computation times created by the usage of rational arithmetic. Note that we used a time-limit of one hour. All times exceeding this limit are denoted by *to*.

First, we will analyze the gain of using ellipsoids to compute the non-autonomous

part of the flowpipe construction instead of using the set representation used for the autonomous part. To test this, we modified the bouncing ball, two tank and cruise control benchmarks by adding an external input that results in an initial bloating by a ball with radius $\beta = 0.0000001$. Computations with different radii had similar results. Unfortunately, the effects on higher dimension could not be tested due to the massive increase of time complexity introduced by external influences. In theory, the effects of using ellipsoids are expected to have an even greater impact on computations on higher dimensions.

We compared the runtimes for different representations, computing the external influence once with the used representation and once with ellipsoids (marked with *). The resulting computation times are shown in Table 4.1. As seen in the table using ellipsoids is an improvement for both $\mathcal{H}$-polytope and support functions. For boxes the usage of ellipsoids has a negative influence which was to be expected due to the simplicity of boxes in comparison to ellipsoids. Focusing on support functions we see that the usage of ellipsoids is significantly improving the computation time reducing it close to the computation time of the corresponding autonomous systems shown later on. Note that for support functions we were using the optimized version with the approaches from Chapter 3. Without those optimizations the computation time for support functions greatly exceeded the time-limit of one hour.

Example	Box	Box*	$\mathcal{H}$	$\mathcal{H}^*$	SF	SF*
Bouncing Ball	**263**	315	326,660	**17,423**	22,188	**2,098**
Cruise Control	**281**	297	656,049	**287,324**	40,241	**6,972**
Two Tank	**173**	206	47,708	**27,346**	175,290	**1,517**

Table 4.1: Comparison of computation times of non-autonomous versions of the benchmarks. The non-autonomous part was once computed with the original representation and once with ellipsoids. The results measured with ellipsoids are marked with *. Times in $[ms]$.

In Chapter 2.2.1 we stated that the external influence has a large impact on the complexity of reachability analysis. Comparing the results from Table 4.1 with the corresponding runtimes of the autonomous versions shown later in Table 4.5 we see that this statement can be observed in our evaluation as well. For boxes the runtimes increased by about 20%. For the more complex representations however, the increase in runtime is significant. Without the usage of ellipsoids the computation times for $\mathcal{H}$-polytopes were between 25 and 200 times longer whereas it was only half as much with usage of ellipsoids. For support functions this effect was even greater. With the usage of ellipsoids we could reduce the computation time to between $\frac{1}{6}$ to $\frac{1}{100}$ times the computation time need with pure support functions. This reduced the increase of necessary time for analyzing non-autonomous systems to 1.4 times the computation time of non-autonomous systems.

In the following we will evaluate the results of the optimizations for support functions as explained in Chapter 3.2.1 and 3.2.2. At the end, we will give a comparison of the runtimes for optimized support functions, $\mathcal{H}$-polytope and boxes.

First, we analyzed the influence of different values for the linear transformation reduction parameter n on the bouncing ball benchmark. As explained in Chapter 3.2.1 this parameter limited the number of successive linear transformations to 2^n before applying the reduction. The results can be seen in Figure 4.7. We see that the choice of n can have a major impact on the computation but unfortunately the runtimes for different n describe have an oscillating behavior. Thus, the choice of an optimal n is a more complex problem than we expected. Further, for some n the linear reduction can even have a negative influence on the computation. On the two tanks benchmark we had a runtime of $1.29 \cdot 10^6 ms$ for $n = 4$, whereas the runtime without optimizations is $1.21 \cdot 10^6 ms$. In conclusion, the gain of the linear reduction is strongly connected to the parameter n and the number of evaluations of the segment which's support function tree is to be reduced.

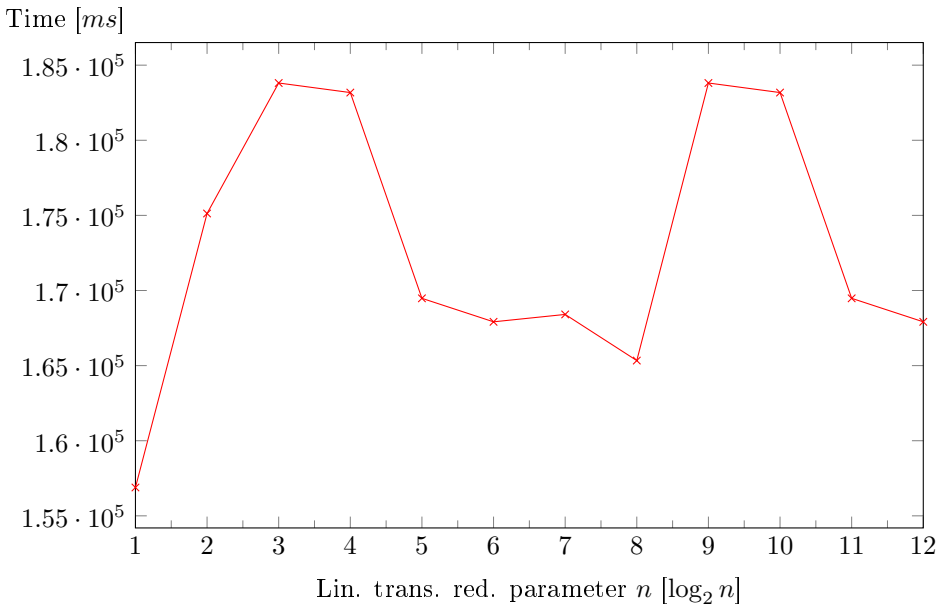


Figure 4.7: Comparison of runtimes for different parameters for linear transformation reduction on the bouncing ball benchmark

The results in Table 4.2 show that both the linear transformation reduction as well as the pre-jump reduction improved the runtime by a small amount. However, comparing the results of the two tanks benchmark with the results on the bouncing ball and cruise control benchmark, we see that there are cases in which the pre-jump reduction does not positively influence the runtime due to the overhead of computing the direct transition matrix.

The usage of the post-jump reduction has shown to be an astonishing improvement for support functions. As seen in Table 4.3 using the post-jump reduction has a significant impact on the computational time improving it by more than a magnitude. With this, we were able to perform reachability analysis on the rod reactor benchmark as well as the switching system benchmark in reasonable time. Further, our proposed

Example	Original sf	Lin. trans. red.	Pre-jump red.
Bouncing Ball	214,106	174,839	**140,035**
Cruise Control	1,244,300	1,073,980	**1,041,940**
Two Tanks	1,214,060	**1,031,010**	1,059,230

Table 4.2: Comparison of runtimes using the linear transformation reduction with $n = 2$ and pre-jump reduction to optimize support functions. Times in $[ms]$.

metric has proven to be efficient. The results with usage of the metric were in all scenarios better than the original support function and also better than an approach where we forced the post-jump reduction after each jump.

Example	Original sf	Post-jump red.	Forced post-jump red.
Bouncing Ball	214,106	**1,369**	1,770
Cruise Control	1,244,300	**5,224**	6,154
Rod Reactor	*to*	**3,070,290**	3,188,840
Switching System	*to*	**1,363,250**	1,629,710
Two Tanks	1,214,060	**1,165**	1,407

Table 4.3: Comparison of runtimes using the post-jump reduction using the introduced metric with a forced post-jump reduction. Times in $[ms]$.

We finally evaluated the combination of post-jump reduction with the other proposed optimizations. However, as seen in Table 4.4 this had worse results than the pure post-jump reduction.

Example	Post-jump red.	Post-jump red. Lin. trans. red.	Post-jump red Pre-jump red.
Bouncing Ball	**1,369**	1,752	1,674
Cruise Control	**5,224**	6,514	6,440
Rod Reactor	**3,070,290**	3,260,940	3,228,830
Switching System	**1,363,250**	*to*	1,619,700
Two Tanks	1,296	**1,165**	1,335

Table 4.4: Comparison of runtimes using the post-jump reduction combined with linear reduction and pre-jump reduction. Times in $[ms]$.

The combination of post-jump reduction with linear reduction is only an improvement if the number of segments between two reductions is high enough as in the two tanks benchmark. Lower numbers of segments only yield an overhead introduced by the linear reduction since the reduction requires a certain number of evaluations to pay off. Furthermore, we tried to combine the pre-jump reduction with the post-jump reduction with an interesting result. The combination of pre- and post-jump reduction also has a negative impact on the computation time. The reason for this is that the computation of a new support function introduced by the post-jump reduction nullifies the impact of the pre-jump reduction except for the creation of the approximating $\mathcal{H}$-polytope, where the costs for the pre-jump reduction are already higher than the

costs of the necessary evaluations for the post-jump reduction. However, there might be cases where applying a pre-jump reduction instead of a post-jump reduction is an improvement. This would be the case for high-dimensional systems with a high number of segments per flowpipe. An approach to solve this conflict between pre- and post-jump reduction is explained later on in Chapter 5.1.

In Table 4.5 we illustrated a comparison of runtime results for different representations. Comparing the optimized support functions with the original implementation we see that the optimizations we proposed had a huge impact on support functions mostly leading to better results as for $\mathcal{H}$-polytopes especially for higher dimensions. However, for the 9-dimensional vehicle platoon all representations exceeded the time-limit due to the complexity additional dimensions introduce. For the 6-dimensional filtered oscillator benchmark all representations failed, except for the box representation. We assume that the high increase in computational time is mainly caused by the drawbacks of exact arithmetic as explained in Chapter 3.1.1. Thus, using floating point arithmetic instead of exact arithmetic might have a drastic effect on the runtime.

Example	d	Box	$\mathcal{H}$	Original sf	Optimized sf
Bouncing Ball	2	235	3,173	214,106	1,369
Cruise Control	3	213	2,920	1,244,300	5,224
Filtered Oscillator	6	3,318	*to*	*to*	*to*
Rod Reactor	3	1,790	*to*	*to*	3,070,290
Switching System	5	34,288	*to*	*to*	1,363,250
Two Tanks	2	156	1,963	1,214,060	1,165
Vehicle Platoon	9	*to*	*to*	*to*	*to*

Table 4.5: Comparison of runtimes for boxes, $\mathcal{H}$-polytopes and support functions with and without optimizations. Times in $[ms]$.

CHAPTER 5
CONCLUSION

Concluding this thesis we have explained the basics of hybrid systems and hybrid automata as well as the flowpipe-based reachability analysis on those systems. Further, we summarized some of the most important state set representations together with their benefits and drawbacks. Then, we explained how to separate the autonomous and non-autonomous part of the reachability analysis by usage of three new sets to reduce the wrapping effect and compute the systems more efficiently. Afterwards, we proposed a new way to compute the non-autonomous system by usage of ellipsoids which has shown to greatly improve the overall runtime for $\mathcal{H}$-polytopes as well as support functions. In addition, we introduced three ways to optimize the computation of autonomous systems using support functions by reducing the complexity of the involved support function tree. Evaluating those optimizations has then shown that all three reductions improve the runtime for support functions. The post-jump reduction has by far the greatest influence with a reduction of the computation time by a magnitude. The other two approaches to improve support functions had a positive influence on the computation as well, even though the effects were quite small. In addition, we saw that there still needs to be time invested in order to join those optimizations and further optimize their influence on the computation. Thus, the computation of flowpipe-based reachability analysis still has a lot of room for improvements and is currently not fitted to precisely analyze high-dimensional systems in reasonable time. Some thoughts on how to optimize our approaches can be found in Chapter 5.1. Unfortunately, those ideas could not be put to test during the creation of this thesis.

5.1 Future Work

Our approaches at improving the flowpipe-computation with support functions has still room for improvements. We have still ideas how to further optimize the three approaches, but we could not put those ideas to the test in this thesis due to the lack of time. Therefore we will leave them to future work and only coarsely describe the ideas here.

The first thing we suggest is using binary search to compute the actual time horizon N' that can be reached in each location within the original time horizon N. This could be realized by computing the set X_N at time N directly from the initial set X_0 and checking if X_N still satisfies the locations invariants. If it does N is a fitting if not, we will continue with the set at time $\frac{N}{2}$ and iteratively search for N'. This will have a major impact on the metric introduced in Chapter 3.2.2 as the reachable time horizon strongly depends on the dynamics and invariants of a location. This improved metric should then be able to determine whether to use the post-jump reduction more precisely. Furthermore, with the actual time horizon we can exactly compute the number of expected segments in the flowpipe. this could be used to create an algorithm to choose the linear transformation parameter n in order to optimize the influence of the linear transformation reduction from Chapter 3.2.1.

Our second suggestion is to analyze the connection between pre-jump and post-jump reduction and find a metric to determine which approach is better fitted for the current jump if a reduction would be useful. This could be realized by computing the flowpipe without reduction up to the point where all segments satisfying a jump guard are known. At this point it should be possible to estimate the costs for the flowpipe generated after the jump with either reduction. If the pre-jump reduction is determined to be less expensive one could re-compute the segments involved in the jump using the pre-jump reduction and otherwise use the post-jump reduction.

BIBLIOGRAPHY

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger, and P. Ho. *Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems*. Springer, 1993.

[ADI03] R. Alur, T. Dang, and F. Ivancic. *Progress on Reachability Analysis of Hybrid Systems Using Predicate Abstraction*. Springer, 2003.

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

[Bor07] H. K. Borgwardt. Average-case analysis of the double description method and the beneath-beyond algorithm. *Discrete & Computational Geometry*, 37(2):175–204, 2007.

[Che15] X. Chen. *Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models*. PhD thesis, RWTH Aachen University, 2015.

[DA01] R. David and H. Alla. On hybrid petri nets. *Discrete Event Dynamic Systems*, 11(1-2):9–40, 2001.

[FGD+11] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, 2011.*, pages 379–395. Springer, 2011.

[GG09] C. Le Guernic and A. Girard. Reachability analysis of hybrid systems using support functions. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, 2009.*, pages 540–554. Springer, 2009.

[GGM06] A. Girard, C. Le Guernic, and O. Maler. Efficient computation of reachable sets of linear time-invariant systems with inputs. In *Hybrid Systems: Computation and Control, 9th International Workshop, HSCC 2006, Santa Barbara, CA, USA, 2006*, pages 257–271. Springer, 2006.

[Gir05] A. Girard. Reachability of uncertain linear systems using zonotopes. In *Hybrid Systems: Computation and Control, 8th International Workshop, HSCC 2005, Zurich, Switzerland, 2005*, pages 291–305. Springer, 2005.

[GK98] P. K. Ghosh and K. V. Kumar. Support function representation of convex bodies, its application in geometric computing, and some related representations. *Computer Vision and Image Understanding*, 72(3):379–403, 1998.

[GT08] S. Gulwani and A. Tiwari. *Constraint-Based Approach for Analysis of Hybrid Systems*. Springer, 2008.

[His01] I. A. Hiskens. Stability of limit cycles in hybrid systems. In *34th Annual Hawaii International Conference on System Sciences (HICSS-34), 2001, Maui, Hawaii, USA*, pages 163–328. IEEE Computer Society, 2001.

[HKPV95] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *27th Annual ACM Symposium on Theory of Computing*, STOC '95, pages 373–382. ACM, 1995.

[KA96] Fukuda K. and David A. First international colloquium on graphs and optimization reverse search for enumeration. *Discrete Applied Mathematics*, 65(1):21 – 46, 1996.

[KV07] A. A. Kurzhanskiy and P. Varaiya. Ellipsoidal techniques for reachability analysis of discrete-time linear systems. *IEEE Transactions on Automatic Control*, 52(1):26–38, 2007.

[KV11] A. A. Kurzhanskiy and P. Varaiya. Reach set computation and control synthesis for discrete-time dynamical systems with disturbances. *Automatica*, 47(7):1414–1426, 2011.

[MMH+11] I. B. Makhlouf, J. P. Maschuw, P. Hänsch, H. Diab, S. Kowalewski, and D. Abel. Safety verification of a cooperative vehicle platoon with uncertain inputs using zonotopes*. *IFAC*, 44(1):9769 – 9774, 2011. 18th {IFAC} World Congress.

[Oeh11] J. Oehlerking. *Decomposition of stability proofs for hybrid systems*. PhD thesis, Carl von Ossietzky University of Oldenburg, 2011.

[ST01] D. Spielman and S. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. In *33rd Annual ACM Symposium on Theory of Computing*, STOC '01, pages 296–305. ACM, 2001.

APPENDIX A

SET OPERATIONS

As we will need a lot of theory on sets, we give an overview of all important operations that are used throughout this thesis.

Definition A.1 (Minkowski sum)
The Minkowski sum is the result of a point-wise addition of two or more sets.

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

Definition A.2 (Union)
Joining together two or more sets

$$A \cup B = \{x \mid x \in A \vee x \in B\}$$

Definition A.3 (Intersection)
Identifying the common part of two or more sets

$$A \cap B = \{x \mid x \in A \wedge x \in B\}$$

Definition A.4 (Convex Hull)
A convex set is a set X such that for all points $x \in X$ it holds

$$x = \sum_{i=1}^{k} \lambda_i x_i, \text{ where all } \lambda_i \geq 0,\, x_i \in X \text{ and } \sum_{i=1}^{k} \lambda_i = 1.$$

The convex hull of a set X is defined analogously as

$$CH(X) = \{x \mid x = \sum_{i} \lambda_i x_i,\, \lambda_i \geq 0,\, x_i \in X \text{ and } \sum_{i} \lambda_i = 1.\}$$

Definition A.5 (Linear Transformation)
A linear transformation with transformation matrix A on a set X is a scaling, rotation, skewing or a combination of those.

$$A \cdot X = \{y \mid \exists x.y = A \cdot x, x \in X\}$$

APPENDIX B

ADDITIONAL INFORMATION FOR BENCHMARKS

In this chapter we give additional information for some benchmarks explained in Chapter 4.1.

B.1 5-Dimensional Linear Switching System

The matrices $A_1, \ldots, A_5$ and $B_1, \ldots, B_5$ are defined as follows

$$A_1 = \begin{pmatrix} -0.8047 & 8.7420 & -2.4591 & -8.2714 & -1.8640 \\ -8.6329 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 1.8302 & 1.9869 & -2.4539 & -1.7726 & -0.7911 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} -0.8316 & 8.7658 & -2.4744 & -8.2608 & -1.9033 \\ -8.6329 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 1.5964 & 2.1936 & -2.5872 & -1.6812 & -1.1324 \end{pmatrix}$$

$$A_3 = \begin{pmatrix} -0.9275 & 8.8628 & -2.5428 & -82329 & -2.0324 \\ -8.6329 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7838 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 0.7635 & 3.0357 & -3.1814 & -1.4388 & -2.2538 \end{pmatrix}$$

$$A_4 = \begin{pmatrix} -1.0145 & 8.9701 & -2.6207 & -8.2199 & -2.1469 \\ -8.6329 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 0.0076 & 3.9682 & -3.8578 & -1.3253 & -3.2477 \end{pmatrix}$$

$$A_5 = \begin{pmatrix} -1.4021 & 10.1647 & -3.3937 & -8.5139 & -2.9602 \\ -8.6329 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ -3.3585 & 14.3426 & -10.5703 & -3.8785 & -10.3111 \end{pmatrix}$$

$$B_1 = \ldots = B_5 = \begin{pmatrix} -0.0845 \\ 0 \\ 0 \\ 0 \\ -0.7342 \end{pmatrix}.$$

B.2 Three-Vehicle Platoon

The matrices A_c, A_n, B_c, B_n are defined as follows

$$A_c = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.6050 & 4.8680 & -3.5754 & -0.8198 & -0.4270 & -0.0450 & -0.1942 & -0.3626 & -0.0946 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0.8718 & 3.8140 & -0.0754 & 1.1936 & 3.6258 & -3.2396 & -0.5950 & 0.1294 & -0.0796 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0.7132 & 3.5730 & -0.0964 & 0.8472 & 3.2568 & -0.0876 & 1.2726 & 3.0720 & -3.1356 \end{pmatrix}$$

$$A_n = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.6050 & 4.8680 & -3.5754 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.1936 & 3.6258 & -3.2396 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0.7132 & 3.5730 & -0.0964 & 0.8472 & 3.2568 & -0.0876 & 1.2726 & 3.0720 & -3.1356 \end{pmatrix}$$

$$B_n = B_c = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T$$

B.3 Cruise Control

The hybrid automaton corresponding to the cruise control benchmark is shown in Figure B.1.

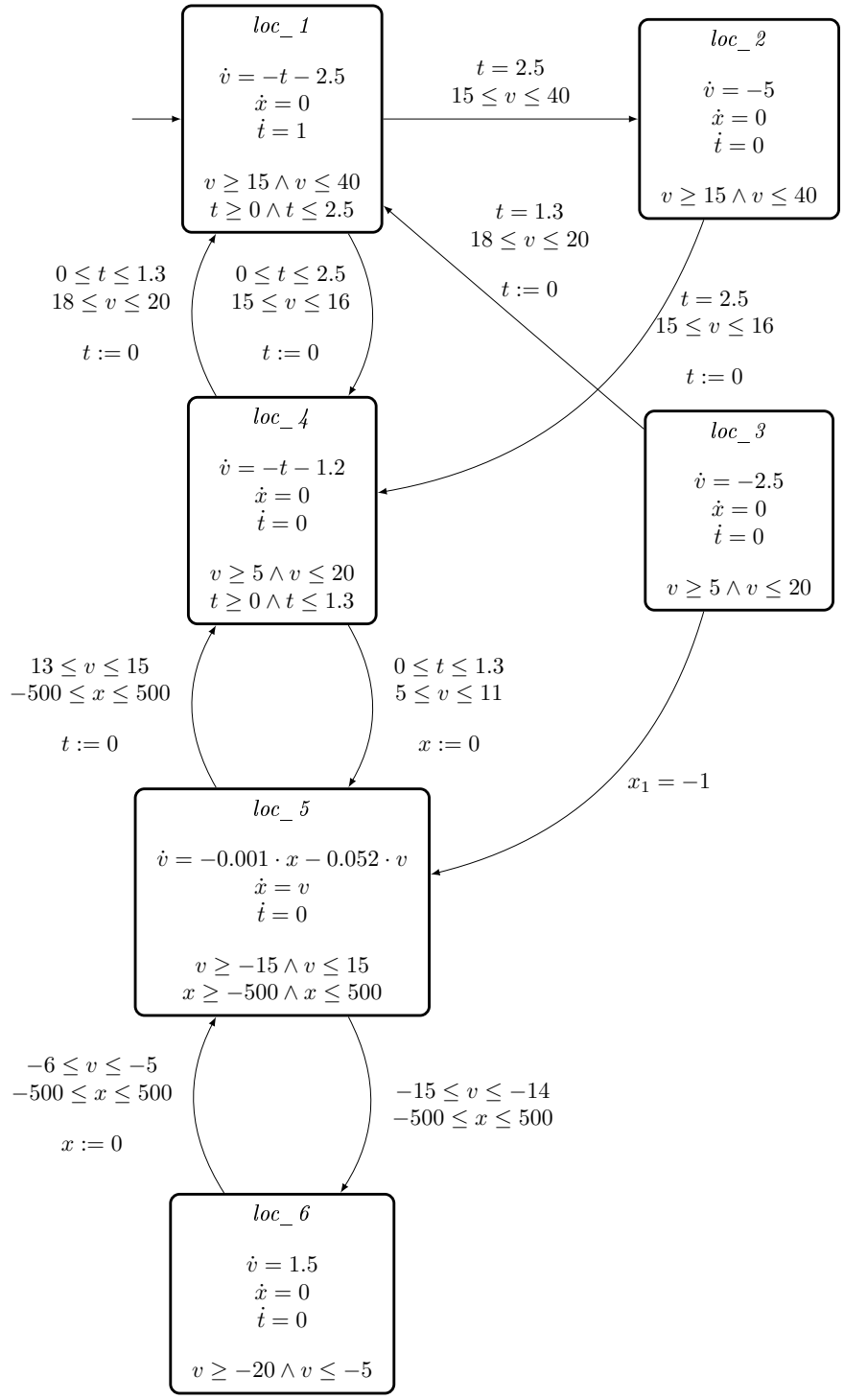


Figure B.1: Hybrid automaton of the cruise control benchmark