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

BACHELOR OF SCIENCE THESIS

COMPUTING SET DIFFERENCE FOR THE REACHABILITY ANALYSIS OF HYBRID SYSTEMS

Kim Amfaldern

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

Additional Advisor:
Dr. Stefan Schupp

Aachen, 01. November, 2021

Abstract

Methods in reachability analysis and verification of hybrid systems have been well-developed over the years, but there are still some challenges to be solved. One of these challenges is the support of the usage of urgent transitions. Urgent transitions force the system to take a jump as soon as such a transition is enabled, so urgent transitions are an alternative to invariants. Most current tools don't support urgent transitions. Here we present a possibility how to treat urgent transitions in the flowpipe-based reachability analysis of hybrid systems by giving an implementation of computing the set-difference in the context of reachability analysis.

Erklärung

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

Kim Amfaldern
Aachen, den 01. November 2021

Acknowledgements

First, I would like to thank Prof. Dr. Erika Ábrahám for giving me the opportunity to write this thesis. She supported me throughout the process of writing this thesis and helped me to develop my theoretical and formal skills. I would also like to thank Prof. Dr.Jürgen Giesl to be the second examiner. Last but not least I would like to thank Dr. Stefan Schupp, who was always helpful in terms of programming and implementing questions. With his help I improved my practical skills in many ways.

Contents

Chapter 1

Introduction

There are many models in computer science, used for different problems. But most of the common models, like transition systems or finite automata, only model discrete behaviour. In this sense hybrid systems are different than most of these models, because hybrid systems model both, discrete and continuous behaviour. Therefore, hybrid sytems are more applicable in processes, that are time-dependent, for example processes, that involve physics.
An example for such a system is a lawn mower as shown in [Her10, ZSR+10]. The lawn mower has to mow the lawn, but may not mow the flowers. For example Figure 1.1 shows a simplified version of the lawn mower. The mower moves with velocity v. The direction it runs can be changed non-deterministically in a 90-degree angle. So if the mower runs out of the lawn field, it will at some point change the direction, but it is not that safety critical, if the mower changes direction right when it leaves the lawn or a few meters later. Therefore, the non-deterministic direction change is sufficient to model this behaviour. In contrast to that the lawn mower should never mow the flowers. Therefore, this is a safety critical behaviour and the mower should change direction before it mows the flowers. This behaviour is easy to model, if we model the lawn mower as a hybrid automaton with urgent transitions, where an urgent transition must be taken, if its guard is enabled. An abstract model of this lawn mower is shown in fig. 1.2, where the name of the state stands for the moving direction of the mower, the urgent transitions are labeled with * and the non-deterministic direction change is represented by the edges labeled with dc.
In this work we want to examine the adaptations, that have to be made in flowpipe-based reachability analysis of hybrid systems, if we consider urgent transitions. The flowpipe-based reachability analysis is a symbolic and iterative method to compute the paths of a system and determine if some safety critical behaviour is possible. It iteratively computes state sets of the system over a given time step to let time evolve. These state sets represent the states, that are reachable in this time step from the step before. In this way the result is a flowpipe of segments representing a path of the system. This method will be explained in more detail in Section 2.4 and **??**. One challenge in the flowpipe-based reachability analysis of the lawn mower lies in the urgent transitions. The characteristics of an urgent transition is, that as soon as it is enabled the transition should be taken and therefore the system should no longer be in the same state. Considering the flowpipe construction, it is possible, that not the complete state set s for one time step satisfies the guard g of an urgent transition, but

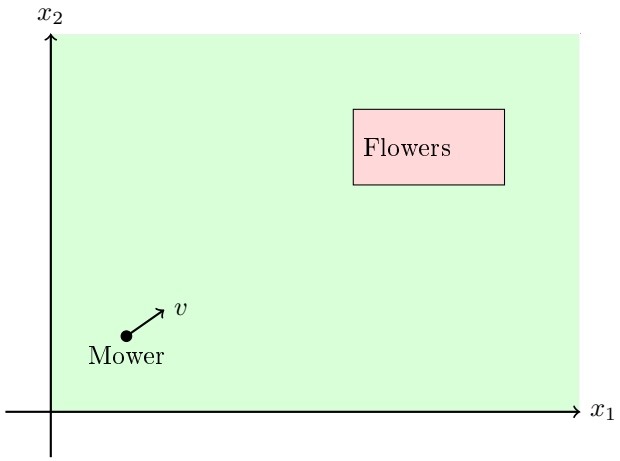

Figure 1.1: Example of a lawn mower
* labels urgent transitions, dc= non-deterministic direction change

a part of it does. Then we do not want, that time evolves from the part satisfying the guard anymore. Therefore, we need to calculate the set difference between the state set and the guard of the urgent transition in order to continue the computation. The computation here has to be split off to the further time evolution of the set $s \setminus g$ and the handling of taking the urgent transition as soon as it is enabled. An example of the reachability analysis for this lawn mower is shown in Chapter 4.
To implement such a reachability anaylysis, we will use the C++ library HyPro. This library offers several state set implementations, conversions between them and already implements a flowpipe-based reachability analysis. This given implementations will be adapted and expanded in order to realize the reachability analysis of hybrid systems with urgent transitions.
To implement such a reachability analysis, that respects urgent transitions, this work first gives some needed theoretical background in Chapter 2. After that we will have a look at some related work of this topic and an algorithm for the set difference, that is needed during the analysis, is introduced in Chapter 3. Chapter 4 then shows how the urgent reachability analysis can be implemented using the HyPro library. At the end in Chapter 5 some benchmarks using urgency are presented.

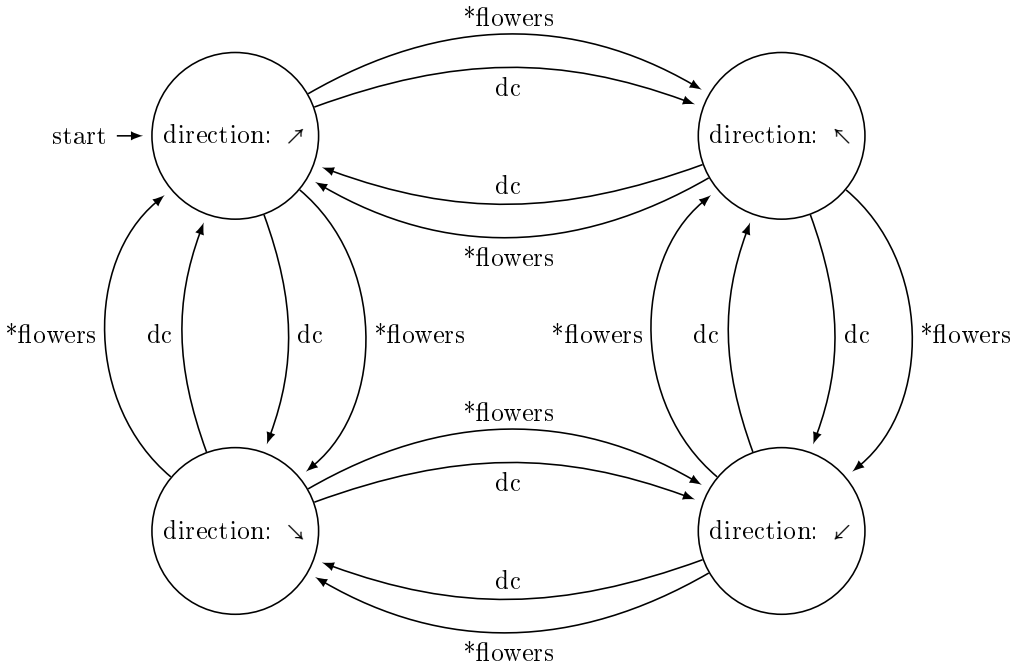

Figure 1.2: Lawn mower automaton

Chapter 2

Preliminaries

Before we implement an algorithm to compute the set-difference, we first need to have a look at some theoretical background.

2.1 Hybrid Systems

As we have seen a hybrid system contains of discrete and continuous behaviour. A popular way to model hybrid systems with such behaviour are hybrid automatons. There are also some common subclasses of general hybrid automaton, that we will have a look at later. First, we will introduce a formal definition. The following definition gives the syntax of a hybrid automaton without urgent transitions.

Definition 2.1.1 (Hybrid Automaton without urgent tarnsitions (Syntax) [SÁC+15, SFÁ19]). *A hybrid automaton is a tuple* $\mathcal{H} = (Loc, Var, Lab, Flow, Inv, Edge, Init)$, *where:*

- *Loc is a finite set of Locations,*
- $Var = \{x_1, \ldots, x_n\}$ *is a finite ordered set of real-values Variables. We also use the vector notation* $x = (x_1, \ldots, x_n)$. *The number* n *is called the dimension of* H. *By* $\dot{Var}$ *we denote the set* $\{\dot{x}_1, \ldots, \dot{x}_n\}$ *of dotted variables (which represent the first derivatives during continuous change), and by* Var' *the set* $\{x'_1, \ldots, x'_n\}$ *of primed variables (which represent values directly after a discrete change). Furthermore,* $Pred_X$ *is the set of all predicates with free variables from* X.
- *Lab is a finite set of synchronisation label (used for the parallel composition of hybrid systems).*
- *Flow:* $Loc \to Pred_{Var \cup \dot{Var}}$ *specifies the dynamic behaviour in each location, the so called flow.*
- *Inv:* $Loc \to Pred_{Var}$ *specifies an invariant for each location.*
- $Edge \subset Loc \times Lab \times Pred_{Var} \times Pred_{Var \cup Var'} \times Loc$ *describes the discrete behaviour of the system.* $(l_1, a, g, r, l_2) \in Edge$ *also is called a jump. For such a jump there exists a discrete transition from* l_1 *to* l_2, *which can be taken if the guard* g *is satisfied. If the transition is taken, its reset function* r *specifies the valuation after this jump.*

- *Init: $Loc \to Pred_{Var}$ assigns to each location an initial predicate.*

A valuation is a function $v : Var \to \mathbb{R}^n$, which assigns a value to each variable in Var. The set of all valuations v in called Val.

To define the behaviour of a hybrid system, there are some more general definitions.
First, we introduce the state of a hybrid automaton. The states of an automaton are pairs (l,v), where $l \in Loc$ is the current location and $v \in Val$ is the current valuation. In order to model the discrete and continuous behaviour, there are discrete and continuous transitions, which can be obtained by the following rules:
Hybrid Automaton (Semantics)[Sch19].

$$\frac{(l,a,g,r,l') \in \text{Edge } v,v' \in \mathbb{R}^n \; v \vDash g \; v' \vDash Inv(l') \; v,v' \vDash r}{(l,v) \xrightarrow{e} (l',v')} \; Rule_{jump}$$

$$\frac{\begin{array}{c} f : [0,\delta] \to \mathbb{R}^n \; \dot{f} : (0,\delta) \to \mathbb{R}^n \; f(0) = v \; f(\delta) = v' \\ l \in Loc \; v,v' \in \mathbb{R}^n \; \forall \epsilon \in (0,\delta) : f(\epsilon) \wedge \dot{f}(\epsilon) \vDash Flow(l) \; \forall \epsilon \in [0,\delta] : f(\epsilon) \vDash Inv(l) \end{array}}{(l,v) \xrightarrow{\delta} (l,v')} \; Rule_{flow}$$

In addition to those general definition, there is one more important point for this work. As we have already seen, we want to work with urgency. Respecting this, we need to add one more component to the hybrid automaton.

Definition 2.1.2 (Hybrid automaton with urgent transitions)**.** *A hybrid automaton with urgent transitions is a tuple $H = (Loc, Var, Lab, Flow, Inv, Edge, Init, Urgent)$, where Loc, Var, Lab, Flow, Inv, Edge, Init are defined like before in Definition 2.1.1 and $Urgent \subset Edge$ is the set of urgent edges of the system.*

With this definition and the semantics of an urgent edge, we can model urgent behaviour.
The semantics of a hybrid automaton with urgent transitions can be modeled in the same way as for automatons without urgent transitions. The difference in this case is the *flow*-rule, which ensures, that no time evolves once an urgent transition is enabled.

$$\frac{\begin{array}{c} f : [0,\delta] \to \mathbb{R}^n \; \dot{f} : (0,\delta) \to \mathbb{R}^n \; f(0) = v \; f(\delta) = v' \\ l \in Loc \; v,v' \in \mathbb{R}^n \; \forall \epsilon \in (0,\delta) : f(\epsilon) \wedge \dot{f}(\epsilon) \vDash Flow(l) \\ \forall \epsilon \in [0,\delta] : (f(\epsilon) \vDash Inv(l) \wedge \neg \exists (l,a,g,r,l') \in \text{Urgent: } f(\epsilon) \vDash g) \end{array}}{(l,v) \xrightarrow{\delta} (l,v')} \; Rule_{flow}$$

To sum up this semantics: an urgent transition is defined by the condition, that as soon as its guard is enabled, the transition has to be taken. So as soon as the jump is enabled, there must no more time evolve in the current state of the system.
In the following we will call hybrid automaton with urgent transitions simply hybrid automaton.
The subclasses of hybrid systems generally differ in the restrictions for the flow and the conditions of the system. Table 2.1 shows such different restrictions.

As this table shows the different restrictions also lead to different results in the decidability of the reachability analysis of the system. In this work we will focus on RA or LHA I and therefore restrict to bounded reachability analysis, as Section 2.4 shows.

Table 2.1: Subclasses of hybrid automata [Sch19]
TA = timed automata, IRA = initialized rectangular automata, RA = rectangular automata, LHA = linear hybrid automata, HA = general hybrid automata
c,c_1,$c_2 \in \mathbb{R}$ are constants, f_{linear} represents linear derivatives, g_{linear} represents linear guard, $\sim \in \{<, >, \leq, \geq\}$

subclass	derivatives	conditions	bounded reachability	unbounded reachability
TA	$\dot{x} = 1$	$x \sim c$	✓	✓
IRA	$\dot{x} \in [c_1; c_2]$	$x \in [c_1; c_2]$, jump resets x when $\dot{x}$ changes	✓	✓
RA	$\dot{x} \in [c_1; c_2]$	$x \in [c_1; c_2]$	✓	×
LHA *I.*	$\dot{x} = c$	$x \sim g_{linear}$	✓	×
LHA *II.*	$\dot{x} = f_{linear}$	$x \sim g_{linear}$	×	×
HA	$\dot{x} = f$	$x \sim g$	×	×

2.2 Intervals

In this work we want to analyze hybrid systems for safety. To do that we will use sets in $\mathbb{R}^n$ to represent the state of a hybrid automaton. In order to do that, we will first have a look at one dimensional sets in $\mathbb{R}$ and some other mathematical background. A convenient way to represent a set is an interval. An Interval is defined by a lower an an upper boundary and contains all values between them, may include them, if it is a closed boundary. Intervals can be defined over every ordered set, but in this work we will define them over the real numbers, $\mathbb{R}$.

Definition 2.2.1 (Interval [SFÁ19]). *An interval $I = \langle l,u \rangle$ is defined by its lower boundary $l \in \mathbb{R} \cup \{-\infty\}$, its upper boundary $u \in \mathbb{R} \cup \{+\infty\}$ and the two boundary types $\langle \in (,[,\ \rangle \in),]$ as the set*

$$I = \{x \mid l \sim_l x \sim_u u\},$$

where for the relation symbols $\sim_l$, $\sim_u \in \{<, \leq\}$ the following holds:

$$\sim_l = \begin{cases} < & \text{if } \langle = (\\ \leq & \text{if } \langle = [\end{cases} \quad , \sim_u \begin{cases} \sim_u = < & \text{if } \rangle =) \\ \sim_u = \leq & \text{if } \rangle =] \end{cases}$$

In addition to this definition a boundary is called closed, if the corresponding relation symbol is $\leq$, where $+\infty$ and $-\infty$ always have to be open boundaries. The function *relation*: $\{(,[,),]\} \to \{<, \leq\}$ maps the relation symbol to the boundary types. In this work we use round brackets (,) to denote open boundaries and squared brackets [,] for closed boundaries, as the definition already shows. Additionally an interval is empty, if the upper boundary is smaller than the lower boundary, as Lemma 2.2.1 shows.

Lemma 2.2.1 (Emptiness of an interval). *An interval $I = \{x \mid l \sim_l x \sim_u u\}$ is empty iff the upper boundary u is smaller than the lower boundary l or equal if at least one relation symbol is $<$.*

$$I = \{x \mid l \sim_l x \sim_u u\} = \varnothing \text{ iff } u < l \text{ or } u = l \wedge (\sim_l = < \vee \sim_u = <).$$

The proof of this lemma follows by the definition of an interval from the set theory. In addition to these definitions there are some more operations on intervals, that will be needed.

Definition 2.2.2 (Intersection of Intervals [Sch19]). *Let $I_1 \langle_1 l_1; u_1 \rangle_1$ and $I_2 = \langle_2 l_2; u_2 \rangle_2$ be two intervals. The intersection $I = I_1 \cap I_2$ is*

$$I = \langle max(l_1,l_2); min(u_1,u_2) \rangle,$$
$$where \langle = \begin{cases} \langle_1 & if\ max(l_1,l_2) = l_1 \\ \langle_2 & if\ max(l_1,l_2) = l_2 \end{cases} and \langle = \begin{cases} \langle_1 & if\ max(l_1,l_2) = l_1 \\ \langle_2 & if\ max(l_1,l_2) = l_2 \end{cases}$$

As Definition 2.2.2 shows the intersection of two non empty intervals can be computed by finding the minimum or maximum of the pairs of lower and upper boundaries. For the set difference of two intervals this is a bit more complex. In general the set difference for two intervals is not only one interval. Therefore, Definition 2.2.3 defines the set difference for two intervals as a set of up to two intervals.

Definition 2.2.3 (Set difference for intervals). *Let $I_1 = \langle_1 l_1; u_1 \rangle_1$ and $I_2 = \langle_2 l_2; u_2 \rangle_2$ be two intervals. The set difference $I = I_1 \setminus I_2$ is the set*

$$I = \begin{cases} \{I_1', I_2'\} \setminus \varnothing & if\ \neg(I_1' = \varnothing \wedge I_2' = \varnothing) \\ \varnothing & otherwise \end{cases}$$

where

$$I_1' = \langle_1 l_1; min(u_1,l_2) \rangle_1',$$
$$I_2' = \langle_2' max(u_2,l_1); u_1 \rangle_1$$

and the choice of $\rangle_1', \langle_2'$ is defined as follows:

$$\rangle_1' = \begin{cases} \rangle_1 & if\ min(u_1,l_2) = u_1 \\ \overline{\langle_2} & if\ min(u_1,l_2) = l_2 \end{cases}, \langle_2' = \begin{cases} \langle_1 & if\ max(u_2,l_1) = l_1 \\ \overline{\rangle_2} & if\ max(u_2,l_1) = u_2 \end{cases}.$$

In this definition $\overline{\langle}, \overline{\rangle}$ means, that the boundary is inverted. This means for the possible boundary types:

$$\overline{(} =]$$
$$\overline{[} =)$$
$$\overline{)} = [$$
$$\overline{]} = ($$

The *min* and *max* function in this definition results from the fact, that two intervals might not intersect. Figure 2.1 shows the different cases, that are possible exemplary for the lower end of the intervals. In this example the type of the boundaries is not considered. Additionally, there are some arithmetical functions on intervals, that will be needed. The first one is the addition of two intervals.

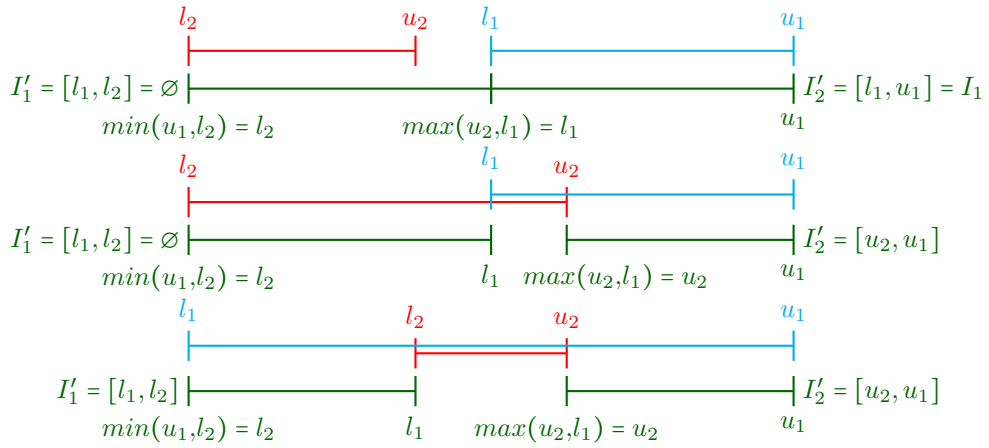

Figure 2.1: Set differnce for intervals.

Definition 2.2.4 (Addition of intervals [Sch19])**.** *Let* $I_1 = \langle_1 l_1; u_1\rangle_1$, $I_2 = \langle_2 l_2; u_2\rangle_2$ *be two intervals. The sum* $I = I_1 + I_2$ *is*

$$I = \langle l_1 + l_2; u_1 + u_2\rangle,$$

where

$$\langle = \begin{cases} (& \text{if } relation(\langle_1) = (\vee relation(\langle_2) = (\\ [& \text{otherwise} \end{cases},$$

$$\rangle = \begin{cases}) & \text{if } relation(\rangle_1) =) \vee relation(\rangle_2) =) \\ ] & \text{otherwise} \end{cases}$$

Note, that we can express a constant $c \in \mathbb{R}$ as a point interval $[c; c]$ and therefore the sum of an interval and a constant simply adds up the constant two both boundaries. Besides addition, we will also use multiplication. This is defined by Definition 2.2.5. We could also express a constant as a point-interval in that case, but since it simplifies the multiplication, the multiplication by a constant is defined seperatly.

Definition 2.2.5 (Multiplication with intervals [Sch19])**.** *Let* $I_1 = \langle_1 l_1; u_1\rangle_1$, $I_2 = \langle_2 l_2; u_2\rangle_2$ *be two intervals and* $c \in \mathbb{R}$ *a constant. Then*

$$I = I_1 \cdot I_2 = \langle min(l_1 \cdot l_2, l_1 \cdot u_2, u_1 \cdot l_2, u_1 \cdot u_2); max(l_1 \cdot l_2, l_1 \cdot u_2, u_1 \cdot l_2, u_1 \cdot u_2)\rangle \text{ is the product of } I_1 \text{ and } I_2,$$

where the boundary type of the result is strict, if at least one of the boundaries of the min/max of the multiplication is strict and

$$I = c \cdot I_1 = \begin{cases} \langle c \cdot l_1; c \cdot u_1\rangle & \text{if } c \geq 0 \\ I = \langle_1 c \cdot u_1; c \cdot l_1\rangle_1 & \text{otherwise} \end{cases} \text{ is the product of } c \text{ and } I_1.$$

To illustrate these definitions we will have a look at a small example.

Example 2.2.1 (Intervaloperations). *Let* $I_1 = [3,7)$ *and* $I_2 = [5;9]$ *be two intervals and* $c = 2$ *a constant.Then we can compute the differnt operations, that were introduced, as follows:*

- *intersection:* $I_1 \cap I_2 = [3,7) \cap [5;9] = \langle max(3,5); min(7,9)\rangle = [5;7)$
- *set differnece:* $I_2 \cap I_1 = [5,9] \setminus [3;7) = \{[5; min(9;3)\rangle, \langle max(7;5); 9]\}$ $= \{[5;3), [7;9]\} = \{\varnothing, [7;9]\} = \{[7;9]\}$
- *addition:* $I_1 + c = [3;7) + 2 = [3;7) + [2;2] = [5;9)$
- *multiplication:* $I_2 \cdot I_1 = [5;9] \cdot [3;7) = \langle min(5\cdot 3, 5\cdot 7, 9\cdot 3, 9\cdot 7);$ $max(5\cdot 3, 5\cdot 7, 9\cdot 3, 9\cdot 7)\rangle = \langle min(15,35,27,63); max(15,35,27,63)\rangle = [15;63)$

2.3 State Set Representation

In the reachability analysis we will use state set representations. State set representations are used to represent the state of a hybrid system in a geometric way. In general all common representations use over-approximation of the state space to define it geometrically. There are different state set representations, e. g. boxes, polytopes, ellipsoids, oriented rectangular hulls, orthogonal polyhedra, template polyhedra and zonotopes. All of them have their advantages and disadvantages. It is always a trade off between precision, meaning less over-approximation, computational effort and memory efficiency.
In this work we will focus on two state set representations for hybrid systems, boxes and polytopes. As we will see, these two representations also focus on different advantages. Since we need to implement such representations and operations on them, we will always define the mathematical set together with a representation of the set, in order to operate on such representations.

2.3.1 Boxes

With the definitions in Section 2.2 we can define a n-dimensional box as a cross product over n intervals.

Definition 2.3.1 (Interval box representation [LG09, Sch19]). *Let* $B_I = (I_1, \dots, I_n)$ *be a vector of* n *intervals. Then* B_I *represents the set*

$$B \subset \mathbb{R}^n, \text{ where } B = \{x \mid x \in I_i \forall i \in \{1, \dots, n\}\}.$$

So B_I *is the interval representation of the box* B*, where* $B_I = \langle l_1; u_1\rangle \times \cdots \times \langle l_n; u_n\rangle$ *is the cross product of* n *intervals.*

The interval definition is one possibility to define a box. Another one is to use two n-dimensional points v_{min}, v_{max} instead of n intervals. One point contains all lower boundaries and the other one all upper boundaries. The Figure 2.2 shows a 2-dimensional box once defined by the intervals $[1;4]$ and $[1;3]$ on the left and once defined by the two points $v_{min} = (1,1)$ and $v_{max} = (4,3)$ on the right.
Regardless of the definition of a box there are $2\cdot n$ values and $2\cdot n$ relation symbols, that have to be stored for a n-dimensional box. For that reason we will mostly use the interval representation in this work. This shows, that boxes are very memory efficient and ensure fast computation. At the same time boxes may lead to a huge

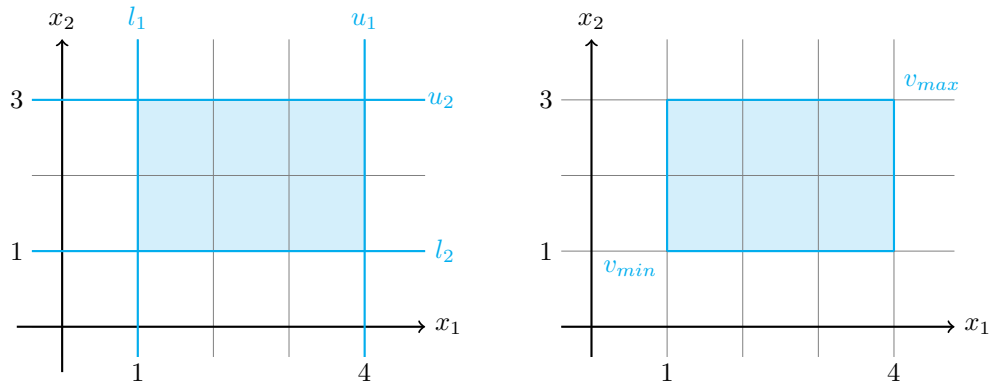

Figure 2.2: Example of a box in 2 dimensions

over-approximation.

There are also some operations on boxes, that will be used in the reachability analysis for hybrid systems. As already mentioned all these operations are defined for boxes by the set theory, but since we want to use them in the implementation, we also have to define them for the representation of a box. The first one is the intersection of two boxes. As Lemma 2.3.1 shows, the intersection of two boxes can be computed in a component-wise manner.

Lemma 2.3.1 (Intersection of boxes [Sch19])**.** *Let $B_1 = I_1 \times \cdots \times I_n$, $B_2 = I'_1 \times \cdots \times I'_n$ be the representations of two boxes. The intersection $B = B_1 \cap B_2$ is the box represented by:*

$$B = (I_1 \cap I'_1) \times \cdots \times (I_n \cap I'_n).$$

The next operation is an affine transformation. This operation transforms a n-dimensional box $B = \{x \mid \forall i \in \{1, \ldots, n\} : x \in I_i\}$ to another box B', where for all $x' \in B$ the equation $x' = A \cdot x + b$ for some $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$ must be satisfied. This also can be done iteratively. The interval representation of the new box B'_I can be computed by using interval multiplication and addition. This algorithm intuitively

Algorithm 1 Affine transformation of a box [Sch19]

Input: interval box representation $B_I = I_1 \times \cdots \times I_n$, matrix $A \in \mathbb{R}^{n \times n}$, vector $b \in \mathbb{R}^n$
Output: interval box representation $B'_I = I'_1 \times \cdots \times I'_n = A \cdot B + b$
for $i = 1, i \leq n, i++$ **do**
  $I'_i = b_i$
  for $j = 1, j \leq n, j++$ **do**
    $I'_i = I'_i + A_{ij} \cdot I_j$
  end for
end for

transforms every boundary of the new interval iterativly.
The last operation, that is mentioned, is the test for emptiness. Since a box is defined

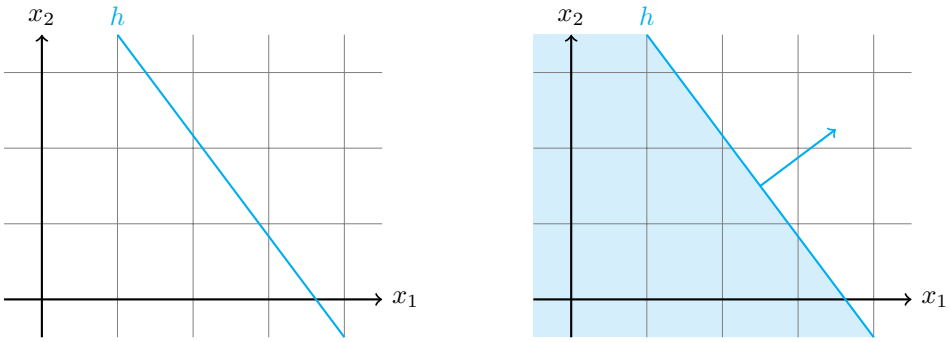

Figure 2.3: Example of hyperplane, halfspace in 2 dimensions

as a cross product of intervals, it gets empty as soon as one interval defining it is empty.

Lemma 2.3.2 (Emptiness of a Box). *A Box $B = I_1 \times \cdots \times I_n$ is empty, denoted as $B = \varnothing$, iff $\exists i \in \{1, \ldots, n\} : I_i = \varnothing$.*

2.3.2 Polyhedra

The second state set representation are polyhedra. Polyhedra will be represented as an intersection of a finite number of half-spaces. Therefore, we will first have a look at some mathematical background, too.

Definition 2.3.2 (Hyperplane [Grü13, Bao05]). *A hyperplane in $\mathbb{R}^n$ is a set of the form*

$$h = \{x \in \mathbb{R}^n \mid a \cdot x = b\}, \text{ where } a \in \mathbb{R}^n,\ a \neq 0,\ b \in \mathbb{R}$$

Definition 2.3.3 (Halfspace [Grü13, Bao05]). *A halfspace in $\mathbb{R}^n$ is a set of the form*

$$H = \{x \in \mathbb{R}^n \mid a \cdot x \leq b\}, \text{ where } a \in \mathbb{R}^n,\ a \neq 0,\ b \in \mathbb{R}.$$

These definitions are illustrated in Figure 2.3. On the left we see the hyperplane h represented by $h = \{x \in \mathbb{R}^2 \mid \frac{4}{3} \cdot x_1 + x_2 = \frac{29}{6}\}$ and on the right there is the half-space $H = \{x \in \mathbb{R}^2 \mid \frac{4}{3} \cdot x_1 + x_2 \leq \frac{29}{6}\}$. h is also called the bounding hyperplane of H.
With this definitions we can formally introduce the definition of a polyhedron.

Definition 2.3.4 (Polyhedron [Grü13, Bao05]). *A polyhedron $P \subset \mathbb{R}^n$ is a convex set given as the intersection of a finite number of closed half-spaces:*

$$P = \bigcap_{i=1}^{m} h_i, \text{ where } m \in \mathbb{N} \text{ is the number of halfspaces defining } P.$$

In addition to polyhedra, we also introduce polytopes, where polytopes are bounded polyhedra, as Definition 2.3.5 shows.

Definition 2.3.5 (Polytope [Grü13, Bao05]). *A polytope in $\mathbb{R}^n$ is a bounded polyhedron and therefore a bounded convex set $P \subset \mathbb{R}^n$ given as an intersection of a finite number of closed half-spaces:*

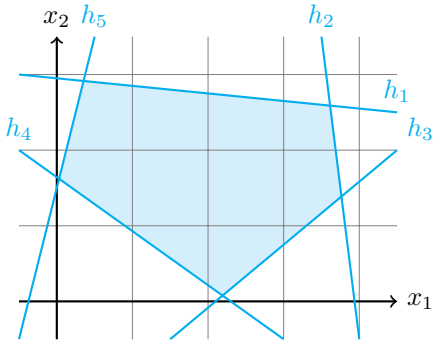

Figure 2.4: Example of a polytope in 2 dimensions

$P = \bigcap_{i=1}^{m} h_i$, *where* $m \in \mathbb{N}$ *is the number of halfspaces defining* P.

In this work we use the standard definitions of convex, bounded and closed sets as it is explained in [Bao05].
With this definitions we can introduce the representation, that is used for polyhedra and polytopes. As both sets are represented by a finite number of half-spaces, these half-spaces can be combined to one matrix and one vector, such that

$$P = \bigcap_{i=1}^{m} h_i = \{x \in \mathbb{R}^n \mid x \in h_i \forall i \in \{1, \dots, m\}\} = \{x \in \mathbb{R}^n \mid A \cdot x \leq b\},$$
$$\text{with } A \in \mathbb{R}^{m \times n},\ b \in \mathbb{R}^m.$$

So the i-th half-space is represented by the i-th row of the matrix A and the i-th component of the vector b.
Figure 2.4 shows a polytope represented by the five half-paces h_i. This representation for polytopes is called $\mathcal{H}$-representation. There is a second representation, the $\mathcal{V}$-representation, where a polytope is represented by its vertices. Both of these two representations have their advantages as disadvantages during computations. Additionally the conversion between these two representations is computationally expensive. In this work we will only use the $\mathcal{H}$-representation, because the set difference operation is easier to compute in this representation.

2.4 Reachability Analysis

The second big part, besides the implementation of the set difference operation, is the integration of the usage of urgent transitions into to reachability analysis. Therefore, we will have an intuitive look at the theoretical background here.
The reachability analysis of a hybrid system is used to compute all reachable states of the system. A state s is reachable, if there exists a path starting in an initial state leading to this state s. To decide, if a certain state is reachable, is undecidable for general hybrid automaton and some subclasses, as Table 2.1 shows.
The goal in the verification of hybrid systems is to check, whether a safety critical state is reachable. The set of all safety critical states is called bad states P_{bad}. In this sense reachability analysis is used to decide, if there is a reachable state, that is bad. Generally, exact calculation are not possible, because the reachability problem for hybrid systems in general is undecidable, therefore there are approaches using either

under- or over-approximation depending on what should be proven. Considering the usage of over-approximation of the state space we can just verify, that a system is safe, if $P_{reach} \cap P_{bad} = \varnothing$. But if $P_{reach} \cap P_{bad} \neq \varnothing$, hence a bad state is reachable, we cannot tell, if the system is unsafe. The reason for that is, that we do not know, if the found bad state really is reachable in the system or if it is the result of the over-approximation. On the other hand the usage of under-approximation can only ensure, that a system is unsafe, if some bad state is reachable, hence $P_{reach} \cap P_{bad} \neq \varnothing$. If no bad state is reached, it can still be the case, that the under-approximation cut away some state, that would lead to an unsafe path. Therefore, under-approximation cannot proof safety.
For this way of reachability analysis, there are two approaches, to check, if a system is safe. The first one is the so called forward reachability analysis. The idea of this method is, to start with a set of initial states and iteratively compute successor states, by letting time elapse or taking discrete jumps. In this way the reachability analysis can stop, once a bad state is reached or a fixed-point is detected, where a fixed-point is a state, that has already been reached in the reachability analysis before and therefore does not have do be considered again. The other approach is called backward reachability analysis. This method starts with the set of bad states and iteratively computes predecessor states. So this analysis can stop once an initial state is reached or a fixed-point is detected. Both of these approaches are not bounded in their runtime, so to ensure the termination of such programs, there is the so called bounded reachability analysis. The bounded reachability analysis determines a fixed number of discrete transitions, that can be taken along one path, such as an upper boundary on the total time duration of a path. These boundaries are called *jump depth* J and *time horizon* T.
The last approach to mention is the flowpipe-construction-based reachability analysis. This approach is based on the bounded reachability analysis. The idea is to divide the *time horizon* T into smaller segments δ, called *time steps*, and calculate the corresponding state set to these segments. All these state sets together build the flowpipe of the system.
There are also other approaches of reachability analysis, for example, using SMT-solving, but in this work we fill focus on flowpipe-construction-based forward reachability analysis.

Chapter 3

Set Difference

Related Work. There are already some theoretical approaches for computing the set difference of polyhedra, like [BMDP02, Bao05] show. The implementation, that is introduced in this work, is also based on these concepts.
Additionally, there are also approaches in integrating urgency in the reachability analysis of hybrid systems. Generally there are two ways to integrate urgency to a hybrid system, urgent locations or urgent transitions. Urgent locations are for example designed by the *tcp (time can pass)* predicate in [NOSY92], that controls the time, that can pass, while the system stays in the same location. This concept is adapted by [SM14] in order to allow no time to pass in a location. This variant of an urgent locations is also similar to an urgent transition without a guard, that is also discussed in [SM14]. More general urgent transitions are discussed in [vBRSR07] in detail. Furthermore, there are approaches focusing on certain subclasses of hybrid automata to integrate urgency. For example, [GV05] shows, how to add urgency to timed automata and discusses the different possibilities in detail. Nevertheless, in most of these researches there is no discrete implementation of urgency in the reachability analysis of hybrid systems.
Moreover, Tristan Ebert is currently writing his master thesis about a CEGAR approach for handling urgency in hybrid systems, which uses the set difference functions, that are introduced here.

After the introduction of the theoretical background, the first main part of this work, the algorithms for the set difference between two boxes or two polyhedra, can be introduced.

3.1 Boxes

The first representation, that is considered, are boxes. As we already mentioned, the goal is to integrate urgency in the reachability analysis of hybrid systems by applying the set difference to a state set and the guard of a transition. Generally the guard of a transition does not have to be a box, but in the following, we assume, that also the guard of a transition is represented by a box.
The basic idea of the algorithm for the set difference is illustrated in Figure 3.1. It consists of the idea to iterate over all dimensions and check, if the interval represent-

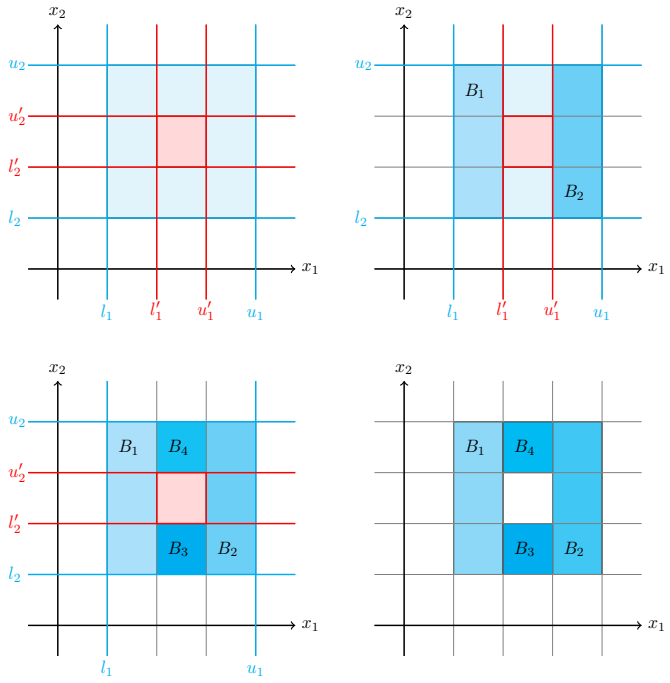

Figure 3.1: Example of the set-difference algorithm for boxes in two dimensions

ing the subtrahend in this dimension is at least partially contained in the interval representing the minuend in this dimension. If that is the case, we can construct one or two new boxes by letting the intervals of the minuend unchanged in all dimensions besides the one we are checking. In this dimension the interval is constructed from the lower bound of the minuend to the lower bound of the subtrahend or from the upper bound of the minuend to the upper bound of the subtrahend. Before continuing with the next dimension also the minuend box has to be adjusted. The adjusting lies in the interval in the considered dimension. This interval has to be changed with respect to which new boxes are constructed. In general, the interval extends from the maximum of the lower bounds of minuend and subtrahend to the minimum of the upper bounds of both.

For the example in fig. 3.1 the minuend is represented by the two intervals $[l_1,u_1] = [1;4]$ in x_1 and $[l_2,u_2] = [1;4]$ in x_2 and the subtrahend is represented by the two intervals $[l_1',u_1'] = [2;3]$ in x_1 and $[l_2',u_2'] = [2;3]$ in x_2. The first dimension, that is considered, is x_1. So the check, if [2,3] is contained in $[1;4]$ is true, and the two boxes B_1 and B_2 are constructed. In addition to these two boxes the minuend box is adjusted in this dimension x_1 to the interval $[2;3]$. With this adjusted box the next dimension, x_2, is considered. The same steps apply and the result of the set difference of the blue minuend box and the red subtrahend box is the vector (B_1,B_2,B_3,B_4) illustrated in the bottom right corner of fig. 3.1. Note, that this solution is not unique. We could also consider the x_2 dimension first and consequently get other boxes. If we would do so, the first two boxes, that are computed, would be $B_1' = [1;4] \times [1;2]$ and $B_2' = [1;4] \times [3;4]$.

So theoretically the set difference for two boxes consists of set differences over intervals and adjusting the box after each set difference. With the knowledge about the set difference for intervals from section 2.2 we can introduce an algorithm, that calculates the set difference for two boxes. This procedure can be done by the algorithm 2,where the function $add(B, B_i)$ adds the box B_i to the vector of boxes B. The check, if a box is empty, is done by iterating over all intervals defining the box and check for each, if it is empty, as shown in section 2.3.1. In this scenario we have to take care of the

Algorithm 2 Set Difference Boxes

Input: box $B_I = [l_1; u_1] \times \cdots \times [l_n; u_n]$, box $G_I = [l_1^{'}; u_1^{'}] \times \cdots \times [l_n^{'}; u_n^{'}]$
Output: vector of boxes $B = B_I \setminus G_I$
$B = \varnothing$
for $i = 1, i \leq n, i++$ **do**
  $B_{low} = [l_1; u_1] \times \cdots \times [l_i; min(u_i, l_i^{'})] \times \cdots \times [l_n; u_n]$
  $B_{up} = [l_1; u_1] \times \cdots \times [max(u_i^{'}, l_i), u_i] \times \cdots \times [l_n; u_n]$
  if $B_{low} \neq \varnothing$ **then**
    $add(B, B_{low})$
    $B_I = [l_1; u_1] \times \cdots \times [min(u_i, l_i^{'}), u_i] \times \cdots \times [l_n; u_n]$
  end if
  if $B_{up} \neq \varnothing$ **then**
    $add(B, B_{up})$
    $B_I = [l_1; u_1] \times \cdots \times [l_i, max(u_i^{'}, l_i)] \times \cdots \times [l_n; u_n]$
  end if
end for
return B

boundary types of the intervals again. We already discovered the bounds that have to be taken by computing the new boxes B_{low} and B_{up}. In addition to that, there are also the boundaries for the adjustment of the minuend box after each dimension. Here the boundary types are just the boundary types of the boundaries that it arises from without any inversion.
In addition to this algorithm depending on the scenario, it might be helpful to check, if the result of the set difference equals the original minuend box or the empty box, to save some computational effort. Only if that is not the case the set difference is computed. This additional check can be done by algorithm 3. If this additional check is performed, we can adapt the set difference computation slightly. Since we know, that we only perform the set difference, if the result is not the original box, it cannot be the case, that both boxes are disjunct and therefore we do not need the *min*,*max* functions anymore. In this case we can just take the corresponding bound of the interval of the subtrahend box instead of the *min*/*max*.

3.2 Polytopes

The second considered representation are polytopes. For polytopes the idea of an algorithm for the set difference is shown in fig. 3.2. In this example the set difference of the blue polytope without the red polytope should be computed. The underlying concept for that is similar to the concept of the algorithm for boxes. The difference is, that instead of changing intervals in a specific dimension to compute a new box,

Algorithm 3 Check for containmend or disjunction

Input: box $B_I = [l_1; u_1] \times \cdots \times [l_n; u_n]$, box $G_I = [l_1^{'}; u_1^{'}] \times \cdots \times [l_n^{'}; u_n^{'}]$
Output: vector of boxes $B = B_I \setminus G_I$
bool empty = true, bool unchanged = false
for $i = 1, i \leq n, i++$ **do**
  if $l_i^{'} \leq l_i$ **then**
    if $u_i^{'} \leq u_i$ **then**
      unchanged = true
    end if
    if $u_i^{'} < u_i$ **then**
      empty = false
    end if
  else
    empty = false
    if $l_i^{'} \geq u_i$ **then**
      unchanged = true
    end if
  end if
end for
if empty **then**
  $add(B, \varnothing)$
  rerunt B
end if
if unchanged **then**
  $add(B, B_I)$
  rerunt B
end if
calculate difference

now half-spaces are added to the original box, to compute a new one. The core of the algorithm is to iterate over all half-spaces defining the subtrahend polytope and compute a new polytope, which is defined by the half-spaces of the minuend polytope and the inverted half-space of the subtrahend polytope. If this polytope is not empty, it is added to the result vector and the considered half-space is added to the minuend polytope. Like before for boxes, the result of the set difference for polytopes, is also not unique. It depends on the order, in which the half-spaces of the subtrahend polytope are considered.

Lemma 3.2.1 (Emptiness of o polytope). *A polytope $P = \{x \in \mathbb{R}^n \mid A \cdot x \leq b\}$ is not empty iff $\exists x \in \mathbb{R}^n : A \cdot x \leq b$. A polytope is empty if such a x does not exist.*

With lemma 3.2.1 checking whether a polytope is empty requires to solve one linear program. In the example in fig. 3.2 the first considered half-space of the subtrahend polytope is $h_1^{'}$. The polytope defined by h_1, h_2, h_3, h_4 and $h_1^{'}$, P_1, is not empty, so it is added to the result vector and the iteration continues with the next half-space, $h_2^{'}$. At the end the five polytopes P_1, P_2, P_3, P_4, P_5 are computed as the result of the set difference blue$\setminus$red.
Although this basic idea is relatively simple, there are some challenges in the calculation of the set-difference for polytopes. The challenges in this approach are depicted

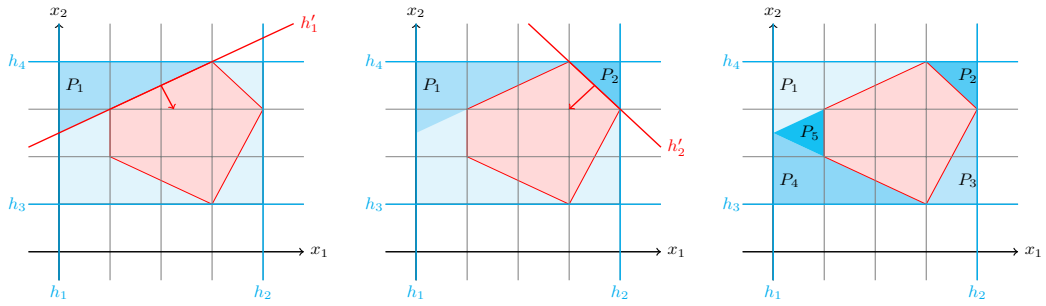

Figure 3.2: Example of the set-difference algorithm for polytopes in two dimensions

in fig. 3.3. The first challenge, depicted on the left, is to identify those half-spaces, defining the polytope G, which do not bound the polytope. These half-spaces are called redundant.

Definition 3.2.1 (Redundancy of a halfspace)**.** *Given a polytope $P = \bigcap_{i=1}^{m} h_i$. Then h_i is redundant, if*

$$P' = \bigcap_{j \in \{1,\dots,m\} \setminus i}^{n} h_j = P.$$

If a polytope only consists of non-redundant half-spaces, it is in minimal representation.

Lemma 3.2.2 (Minimal representation [Bao05])**.** *A polytope $P = \bigcap_{i=1}^{m} h_i$ is in minimal representation iff the removal of any half-space h_i would change P.*

With lemma 3.2.2 redundant half-spaces are those, that can be removed, without changing the polytope. These half-spaces should not be considered in computing the set difference, as they may lead to an avoidable overhead. In order to do that, we will make sure, that both polytopes are in minimal representation.

Lemma 3.2.3 (Calculation Redundancy of a Halfspace)**.** *Given a polytope $P = \bigcap_{i=1}^{m} h_i$, represented by $P = \{x \in \mathbb{R}^n \mid A \cdot x \leq b\}$. Checking whether h_i is redundant can be done by checking if it exists a x such that:*

(1) $A' \cdot x \leq b'$, where A', b' are obtained from A,b by removing the i-th row
(2) $-A_i \cdot x < -b_i$

If no such x exists, h_i is redundant.

If there exists a x such that (1) and (2) are satisfiable, then h_i is not redundant. According to lemma 3.2.3 this can be done by solving one linear program for every halfspace. For example, in fig. 3.3 the half-spaces h'_6 and h'_7 are removable.
The second challenge is, to decide, which half-spaces of G intersect with P. This is helpful, because half-spaces of G, that do not intersect with P, do not have an effect on the set-difference $P \setminus G$. In fig. 3.3 the half-spaces h'_2 and h'_3 in the picture in the middle do not intersect with P, so those do not have to be considered in computing the set-difference. Only the half-spaces h'_1, h'_4 and h'_5, which we will call active bounds of G, will be considered in the iteration over the half-spaces of G.

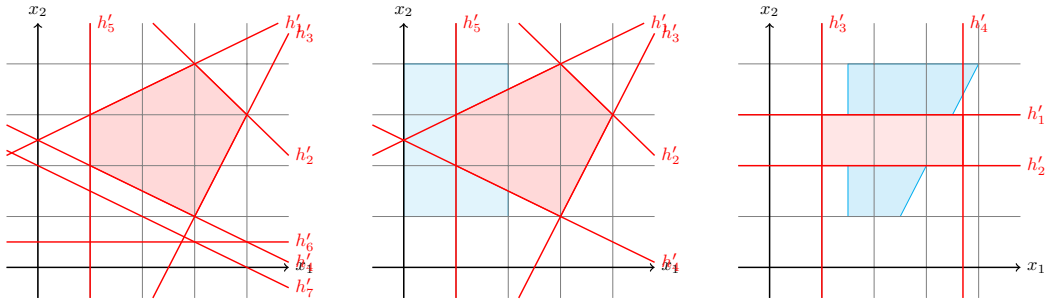

Figure 3.3: Challenges set-minus polytopes

Lemma 3.2.4 (Active bounds). *Given the two polytopes* $P = \bigcap_{i=1}^{m} h_i$*, represented by* $P = \{x \in \mathbb{R}^n \mid A \cdot x \leq b\}$ *and* $G = \bigcap_{i=1}^{k} h'_i$*, represented by* $G = \{x \in \mathbb{R}^n \mid A' \cdot x \leq b'\}$ *in minimal representation. A half-space* h'_i *of* G *is called active iff* $\exists x$ *such that:*

$$(1)\ -A'_i \cdot x \leq -b'_i \wedge A \cdot x < b$$
$$(2)\ A' \cdot x \leq b'$$

The third challenge is shown on the right in fig. 3.3. Here the red polytope G is already in minimal representation, additionally we can identify h'_3 as not active. The challenge in this case lies in the half-space h'_4. This half-space intersects with P, but the set-difference can (and should) be computed without it. Therefore, h'_4 should also not be an active bound. This is the reason for the constraint (2) in lemma 3.2.4. This constraint ensures, that there is a point x that lies on the facet of G which is part of h'_i, in this case h'_4. While the (1) constraint in lemma 3.2.4 makes sure, that the intersection of the inverted considered half-space an the polytope P is not empty and even not just a hyperplane or a point. Again for the decision if a half-space is active or not one linear program has to be solved for every halfspace.

The algorithm 4 shows, how the set difference for two polytopes can be computed, by representing the polytopes as sets. This is an intuitively easy to understand possibility to comprehend the idea of the algorithm. But because we also want to implement the algorithm later (chapter 4), the algorithm 5 shows an implementation in pseudo code, where a polytope represented by a pair of a matrix A and a vector b, such that the polytopes contains all points x which satisfy $A \cdot x \leq b$.

In this algorithm the function get_active_bounds(G, P) returns the active bounds of G with respect to P according to lemma 3.2.4. Then it iterates over all active bounds and constructs a new polytope P_i. The function *addrow*() adds a new row to the according matrix/vector. So that the two *addrow*()-operations on A'_p and b'_p together represent the addition of a new half-space to the (A,b) representation of a polytope. The same principle later applies for P. Here P is adjusted with respect to the considered active bound, if the computed P_i is part of the result, i. e. is not empty.

Algorithm 4 SetMinus Polytopes - set representation

Input: fulldimensional polytopes $P = \{x \in \mathbb{R}^n \mid P^x \cdot x \leq P^c\}$, $G = \{x \in \mathbb{R}^n \mid G^x \cdot x \leq G^c\}$ in minimal representation
Output: polytope- collection $P \setminus G$
G_a = {active bounds of G}
if $G_a = \varnothing$ **then**
  return $\varnothing$
else
  for all bounds $g_i \in G_a$ **do**
    $P_i = P \cap \{x \in | \; g_i^x \cdot x \geq g_i^c\}$
    if $P_i \neq \varnothing$ **then**
      $P_S = P_S \cup P_i$
      $P = P \cap \{x \in | \; g_i^x \cdot x \leq g_i^c\}$
    end if
  end for
  return P_S
end if

Algorithm 5 SetMinus Polytopes- (A,b)-tupel representation

Input: representations of fulldimensional polytopes $P = \{x \in \mathbb{R}^n \mid A_p \cdot x \leq b_p\}$: $P = (A_p, b_p)$, $G = \{x \in \mathbb{R}^n \mid A_g \cdot x \leq b_g\}$: $G = (A_g, b_g)$ in minimal representation
Output: set of polytopes $P \setminus G$
G_a = get_active_bounds(G, P)
if $G_a = \varnothing$ **then**
  return $\varnothing$
else
  for all rows $(A_{g,i}, b_{g,i} \in G_a)$ **do**
    $(\overline{A_{g,i}}, \overline{b_{g,i}}) = getInverse((A_{g,i}, b_{g,i}))$
    $A'_{p,i} = addrow(A_p, \overline{A_{g,i}})$
    $b'_{p,i} = addrow(b_p, \overline{b_{g,i}})$
    $P_i = (A'_{p,i}, b'_{p,i})$
    if $P_i \neq \varnothing$ **then**
      $P_S = P_S \cup P_i$
      $A'_p = addrow(A_p, A_{g,i})$
      $b'_p = addrow(b_p, b_{g,i})$
      $P = (A'_p, b'_i)$
    end if
  end for
  return P_S
end if

Chapter 4

HyPro

The C++ library HyPro offers several state set representations for hybrid systems. These representations can be used for reachability analysis of hybrid systems, especially for flowpipe construction based algorithms. HyPro also offers operations on these representations and conversion methods between them. Additionaly, there are some more tools provided by HyPro, as fig. 4.1 shows. This work shows an algorithm, that implements the set difference for two of the representations used in HyPro. In addition, it provides a method which supports urgent transitions in hybrid systems by using the set difference methods in the flowpipe-based reachability analysis of these systems.

4.1 General Reachability Analysis

As mentioned in chapter 2 we will focus on flowpipe-based forward reachability analysis. So the reachability analysis is bounded in the number of discrete and continuous transitions, that can be taken. In HyPro these bounds can be set by so called *AnalysisParameters*. The general concept of the reachability analysis in HyPro works as follows. There is an analyser, a worker and some handlers, where the analyser coordinates the reachability analysis. It checks, if the jump depth is reached or a fixed-point is detected. If there is no reason to stop the analysis, the analyser creates tasks for the worker. These tasks are for example computing the time evolution of the system in a given state or computing the jump successors in a given state. During this computations the worker and the analyser need to check and perform various steps, for example, checking the invariant of a location, applying the reset for a discrete transition or checking if a bad state is reached. For each of these tasks there is one handler, which performs the needed tasks.
This concept uses a *ReachTree* to coordinate the work. The *ReachTree* basically stores the paths of the system together with their flowpipe in each location. So the *ReachTree* starts with just a root, which is the inital configuration and every time a discrete transition is taken a child node is added.
The general implementation of this concept is shown in algorithm 6. As this algorithm shows, the basic concept is, to calculate all time successor states in the current location, which leads to a flowpipe of time segments and then check for every flowpipe segment, if the guard of a discrete transition is enabled. Remembering the *ReachTree* concept this way of computing jump successors may lead to a strong branching in the

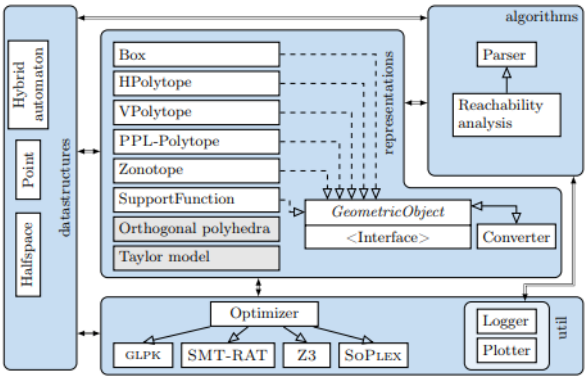

Figure 4.1: Organizational structure of HyPro [SAMK17]

Algorithm 6 Analyser

```
while there exists unprocessed ReachTreeNode do
    currentNode = nextNode()
    currentNode.computeTimeSuccessors()
    if badstatereached then
        return UNSAFE
    end if
    if !fixedPointDetected then
        currentNode.computeJumpSuccessors()
        for all JumpSuccessor do
            addNewNode();
        end for
    end if
end while
```

tree, because for every flowpipe segment satisfying the guard of a discrete transition, a new Node is added to the *ReachTree*. In order to somehow control the branching, there a concepts like *Aggregation* or *Clustering* which are discussed in chapter 4. But first we need to understand the general concept. As algorithm 6 shows, we need to compute time- and jump successors and check for bad states or fixed points.

4.1.1 General TimeSuccessor Computing

Generally the time successor computation has to let time evolve until the time horizon is reached. This computation can be done by applying an affine transformation to the state set, since we consider linear systems in this work. The flow of linear systems can be expressed by an equation of the form $x' = A \cdot x + b$.

Example 4.1.1. *For example, let there be two variables in the system,* x_1, x_2. *We want* x_1 *to have a constant change* $\dot{x_1} = 5$ *over time and* x_2 *a linear change depending on* x_1, $\dot{x_2} = x_1$. *Then we can specify this flow by the matrix* $A = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}$ *and the vector*

$b = \begin{pmatrix} 5 \\ 0 \end{pmatrix}$ *And consequently get the intended flow of the system by the equation*

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = A \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + b = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \begin{pmatrix} 5 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ x_1 \end{pmatrix}$$

Note that we could also use equations of the form $x' = A \cdot x$ without the addition with a vector b, if we add an extra dimension with zero flow to represent the constant flow, as in [Sch19]. But since we already know how to apply an affine transformation to the considered state set representations, we stay with the above definition. Before this time evolution even starts there is one more task to handle, that is shown in algorithm 7. We have to compute the first segments, that is the starting point for the flowpipe construction. We will not go into detail how this computation works, but basically we have to make sure, that at every time, the actual reachable states are contained in the state set, that is computed by the time evolution. Because the time evolution just computes the state after a fixed time step, we do not know for sure, that in between these two time points, we have not left the set of the actual reachable states. Therefore, some bloating is applied to the first computed segment.
In addition to that during the time evolution computation, two other properties must be checked, (1) the invariant of the current location may not be violated, (2) a bad state should not be reached.
With lemma 4.1.1 (1) can be done by intersecting every new time segment with the invariant of the system and only continue the computation, if the intersection is not empty.

Lemma 4.1.1 (Satisfying a Invariant)**.** *A segment s does not fulfill the invariant of a location loc, if $s \cap inv(loc) = \varnothing$*

For (2) the segment has to be intersected with the *bad states* and if this intersection is not empty a bad state has been reached and the computation can stop. With this knowledge algorithm 7 shows how such a *computeTimeSuccessor()-method* would look like. In this algorithm new segments are computed until the time bound is reached, no part of the segment fulfills the invariant or a bad state is reached. So after this algorithm the flowpipe for the current *ReachTree*-Node is computed.

4.1.2 General Jump Successor Computation

As we have already mentioned before, the jump successor computation generally checks for every flowpipe segment, if it at least partially satisfies a guard of a discrete transition. If a guard is enabled the jump should be taken to continue the reachability analysis. Now there is one more concept to control the branching in the *ReachTree*, that arises from this working concept.
It might be the case, that during the flowpipe construction a number of segments, that all enable the same discrete transition, is computed. In order to reduce the number of new *ReachTree*-Nodes, that are created, these segments can be combined to larger segments. This concept is called *Aggregation* or *Clustering*. Both approaches take several segments of the flowpipe, that all satisfy the same guard and unify them to a larger segment. The difference is, that *Aggregation* takes all segments, that satisfy one guard and unifies it to one new segment, while *Clustering* uses a fixed number of segments, that are unified [Sch19]. Figure 4.2 shows no aggregation on the left,

Algorithm 7 ComputeTimeSuccessors-function

```
firstSegment = constructFirstSegment();
if firstSegment ∩ loc.getInvariant() != ∅ then continue
end if
if firstSegment ∩ badStates != ∅ then
    return UNKNOWN
else
    flowpipe.add(firstSegment)
end if
while !TimeBoundReached do
    segment = applyTimeEvolution()
    if segment ∩ loc.getInvariant() == ∅ then
        return SAFE
    end if
    segment = segment ∩ loc.getInvariant()
    if segment ∩ badStates != ∅ then
        return UNKNOWN
    end if
    flowpipe.add(segment)
end while
```

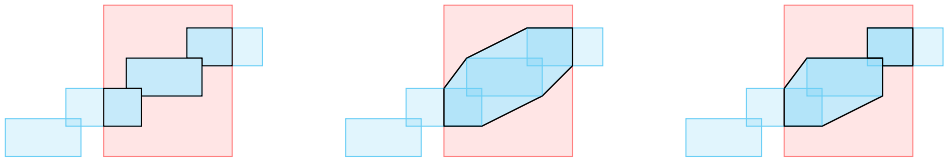

Figure 4.2: Controlling the ReachTree Branching: Aggregation vs Clustering

aggregation in the middle and clustering on the right.

With this knowledge algorithm 8 shows, how the TimeSuccessor Computation can be implemented. This algorithm first computes pairs of a transition and flowpipe segments, that satisfy this transition with the function *getenabledsegments*. This computation can be done by iterating over all transitions and segments and check, if the intersection of the guard of the transition with the segment is not empty. When these pairs are computed, they can be used to compute the aggregation or clustering by building the union of the segments, that should be unified. This is done in *applyAggregation*. With this new segment or new segments, if clustering is applied, the reset of the considered transition can be applied to the new segment and a new node is added to the *ReachTree*.

4.2 Urgent Reachibility Analysis

After introducing the general concept of the reachability analysis implementation, we now want to add the possibility of urgent transitions to it. For this reason we will

Algorithm 8 ComputeJumpSuccessors-function

```
Input: ReachTreeNode: location loc, flowpipe of this location flowpipe
vector (vector(segments), transition) enabledsegments
enabledsegments = getenabledsegments(flowpipe, loc)
for all (segments, transition) ∈ enabledsegments do
    vector newsegments = applyAggregation(segments, transition)
    for all newsegment ∈ newsegments do
        resetedsegment = applyReset(newsegment, transition)
        addNode(resetedsegment, transition)
    end for
end for
```

again have a look at the time- and jump successor computation.

4.2.1 Urgent TimeSuccessor Computing

When using urgent transition the algorithm 7 we have seen before has to be adapted in the following way. The characteristic of urgent transitions is, that the transition has to be taken as soon as it is enabled. Therefore, the control should no longer stand in the current state once an urgent transition is enabled. The problem here comes from the flowpipe construction. One time step has a chosen, but fixed size. So at one point it might be the case, that a time segment only partially satisfies the guard of a transition. At this point this segment has to be adapted. Because the urgent transition has to be taken has soon as it is enabled and not sometime later, we must make sure, that at the *timesuccessor*-computation not the whole segment is mentioned. For the computation of the time successors this means, that during the computation there must be a set difference operation subtracting the guard of an enabled urgent transition from the time segment. Such an algorithm must (1) iterate over all urgent transitions of the current location and check for each, if it is enabled and (2) if a transition is enabled compute the set difference of the time segment with the guard of this transition.
(1) Can be done by computing the intersection of the time segment and the guard of the respective transition. If this intersection is not empty, the guard is enabled. If an enabled urgent transition is found in (1), (2) can be done by computing the set difference of the segment and the guard using the algorithms shown in chapter 3. Algorithm 9 shows a pseudo code algorithm, that focus on the urgent computation and does not mention other checks, like for the invariant or bad states. A detailed algorithm is shown in Algorithm 10, which is an implementation in HyPro.

Compared to algorithm 7 this implementation has the same structure. The difference is the one additional check, if an urgent transition is enabled. If that is the case the result is computed by using the *SetMinus*-function. Because the result of this set difference in general is not only one segment, all resulting segments have to be mentioned in the further computation. That is the reason for the *segments*-vector used in this algorithm. This vector stores all segments, that are computed during the set difference operation. In addition to that, we also have to consider, that there is maybe more than one urgent transition. The existence of multiple urgent transition leads to additional computational effort. All segments, that are the result of the set difference with one urgent transition, have to be checked by the next urgent transi-

Algorithm 9 ComputeTimeSuccessors-function using urgent transitions

```
firstSegment = constructFirstSegment()
vector segments = urgencyCheck(firstSegment)
addToFlowpipe(segments)
while !TimeBoundReached do
    for all seg ∈ segments do
        seg = applyTimeevolution(seg)
        vector tmpsegments = urgencyCheck(seg)
    end for
    segments.clear()
    for all tmpseg ∈ tmpsegments do
        if tmpseg ! = ∅ then
            segments.add(tmpseg)
            addToFlowpipe(tmpseg)
        end if
    end for
end while
```

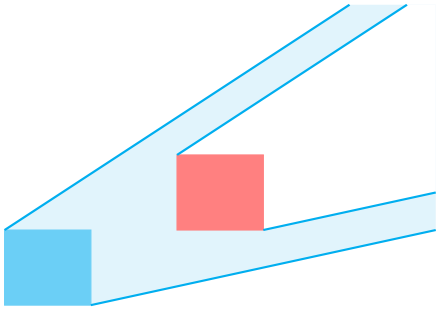

Figure 4.3: Example of an urgent guard casting a shadow

tion again. For this reason there is the *tmpsegmets*-vector. This vector keeps track of the computation for one segment during the urgent transition check and adds all non empty segments to the *segments*-vector afterwards. To keep track of these both vectors there are lower and upper bounds storing the indices needed for the computation.
Additionally, there is one more point to mention for the time successor computation supporting urgent transition. The guard of a transition casts a shadow to the segment, which represents the parts of the segment, that would not be reached, if the urgent transition has been taken as soon as it is enabled. Figure 4.3 shows such a case. In this example the flow is directed left to right and bottom up. Therefore, the red urgent guard forces the system to leave the current state on the left and the lower side of the guard. So the white space would not be reached, because the system left the state before. This leads to a problem in the reachability analysis supporting urgent edges, since the set difference of the timeevelotion of the blue set with the red guard set would compute a result that includes the white space, that should not be reached.

4.2.2 Urgent Jump Successor Computation

Our implementation of the computation of the jump successors for a flowpipe in a location does not differ much from the general implementation. Only the aggregation step would have to be adapted, but this is not part of this work.
The reason, that there is no real need to adapt the computation is the usage of over-approximation. As we have seen in chapter 2 there is much effort for boxes to keep track of the bound types and for polytopes we do not even use strict inequalities. This is why the set difference operations used in the time successor computation over-approximate the real difference. The over-approximation there lies in the borders of the boxes and polytopes. For example, in fig. 2.1 we discussed the theoretical result of the set difference for intervals, which can be used in a component wise manner to compute the set difference for boxes. Although in the implementation we do not use this exact computation, since we only consider the values of the bounds for the result of the set difference but not the bound type. We only use weak inequalities for the minuend, the subtrahend and the difference. This leads to a useful advantage in the jump successor computation. Since an urgent transition should be taken as soon as it is enabled, for linear hybrid systems this means, that the jump should be taken at the borders of the guard. Exactly these borders are still contained in the flowpipe of the system, so that there is no additional effort for the jump successor computation. Of course we could also do exacter computations by using the interval set difference for boxes and a symbolic representation for polytopes, but this would lead to additional effort during the whole reachability analysis process. We would have to store the segments intersecting urgent guards in the time successor computation and remember them for the jump successor computation, where then exactly the first time point, that the guard is enabled, has to be computed in order to apply the reset. This would again lead to another problem, since we can only determine that for linear systems, with weak guards.

Chapter 5

Benchmarks

In this chapter we want to have a look at some hybrid systems, that use urgent transitions. We will analyse the relevance of the support of urgent transitions and the advantages and disadvantages, that it brings. To do so, we will apply the flowpipe-based forward reachability analysis, that is introduced in Chapter 4 without aggregation or clustering. We will have a look at two different systems. For the first one we will use boxes as state set representation and for the second one polytopes.

5.1 Lawn Mower

The first system is the lawn mower, that we already introduced. The automaton representing the system, that is discussed is shown in Figure 5.1. This automaton is slightly different than the one before. In this automaton the direction can not change non deterministicly, but when the mower leaves the lawn or before it mows the flowers. The safety critical behaviour of this system is the lawn mower mowing the flowers. So regardless of the location the system is in, the valuation should never satisfy the condition $30 > x > 70 \wedge 100 > y > 150$. Therefore, the transitions to ensure, that such behaviour does not happen, are urgent. Indeed, this urgency ensures safety of that system. Without the transitions being urgent the system is no longer safe, because the system can take these transitions as soon as they are enabled, but without urgency, it does not have to. It can stay longer in the location and enter the bad states by letting time elapse. In conclusion the urgency in this automaton ensures safety of the system, but we also want to have a look at another automaton modeling a similar safe system without urgency.

This automaton splits the lawn field in nine parts, like Figure 5.2 shows. The flowers build the middle part and around the flowers, there are eight field parts that have to be mowed. For each of these parts there are four locations, that manage the direction the mower is going, like before. There are also transitions like before to control, that the mower does not leave the field and does not mow the flowers, but additionally there are transitions that manage the switch between the field parts. A model of this automaton can be found in the HyPro library.

If we now compare the two safe automatons of the lawn mower, we notice, that the time of computations, where the mower mows the same route in both automatons, meaning the same number of direction changes is applied, is not that different, as Table 5.1 shows. The reason for that is, that the urgent automaton introduces more

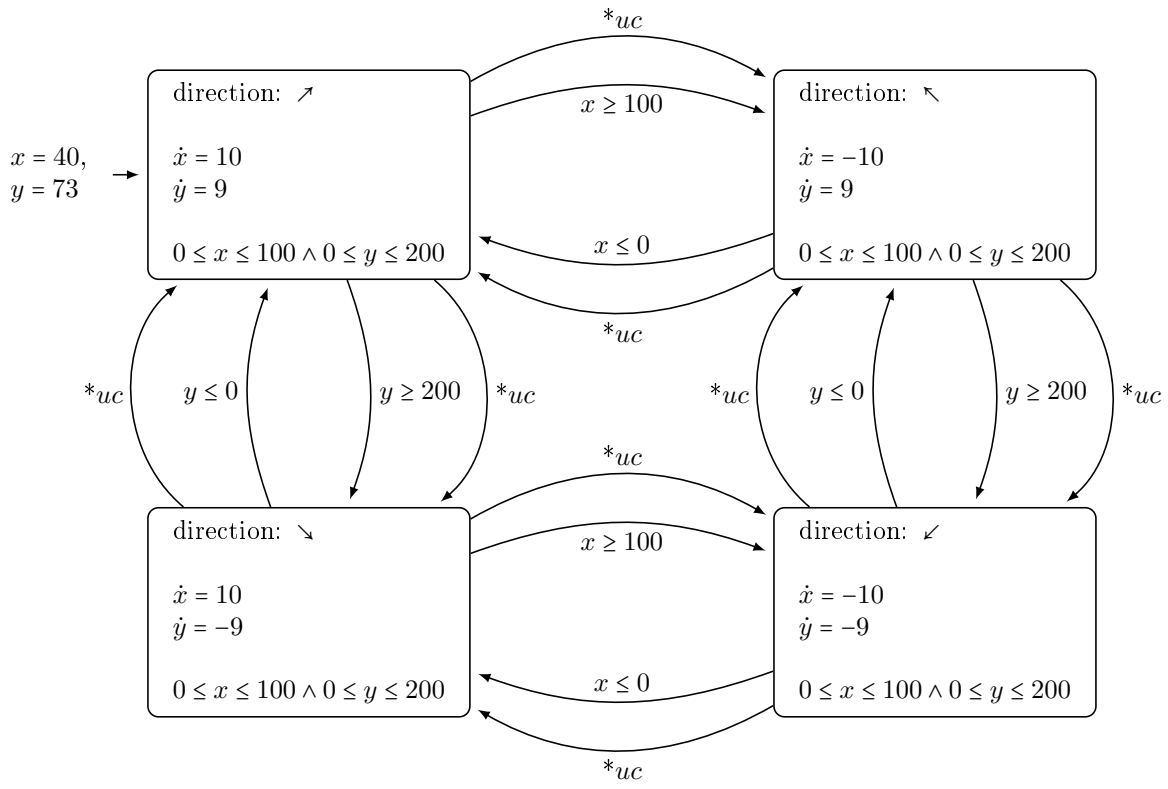

Figure 5.1: Lawn mower automaton .
* marks a transitions as urgent, *uc* stands for the urgent condition $30 \le x \le 70 \wedge 100 \le y \le 150$

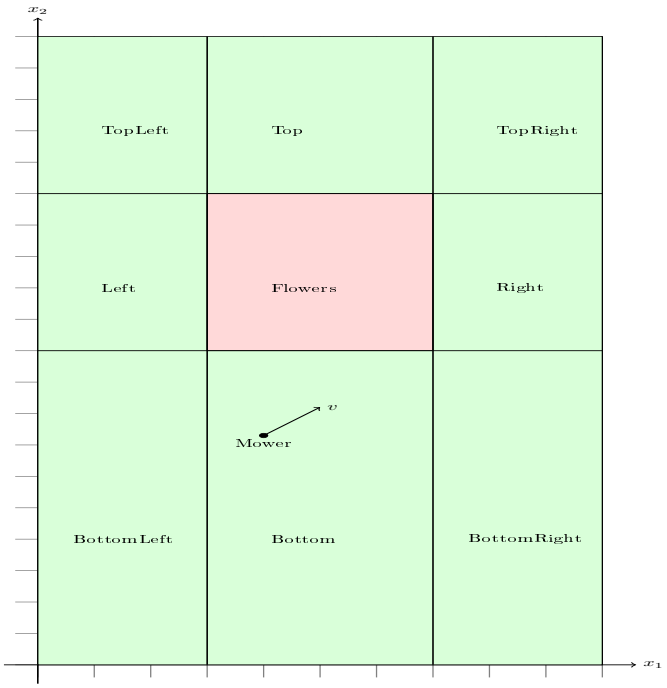

Figure 5.2: Lawn/Flower Field

Table 5.1: Comparison Urgency vs. No-Urgency, Time in Seconds

Number of Direction Changes	3	5	7	10
Urgent Automaton	0.015	0.046	0.186	0.424
Non-Urgent Automaton	0.017	0.052	0.110	0.295

overhead in handling the urgent transitions, meaning checking, if they are enabled and computing the set difference if that is the case, but the other automaton consists of far more states, which leads to much more effort in handling the ReachTree of the automaton, because it consists of much more nodes. If we have a closer look at these times we notice, that the urgent automaton is faster at first, but gets much slower the more transitions are taken. The reason here, is that the set difference computation, that is used for the urgency handling, costs more computational effort, that the intersect computation, that is used without urgency.
Independently from the time efficiency of these two automatons we have to mention, that the second automaton without urgency only works in this case, because we have constant derivatives and always ensure, that we enter a location with a valuation, that satisfies the invariant of this location.

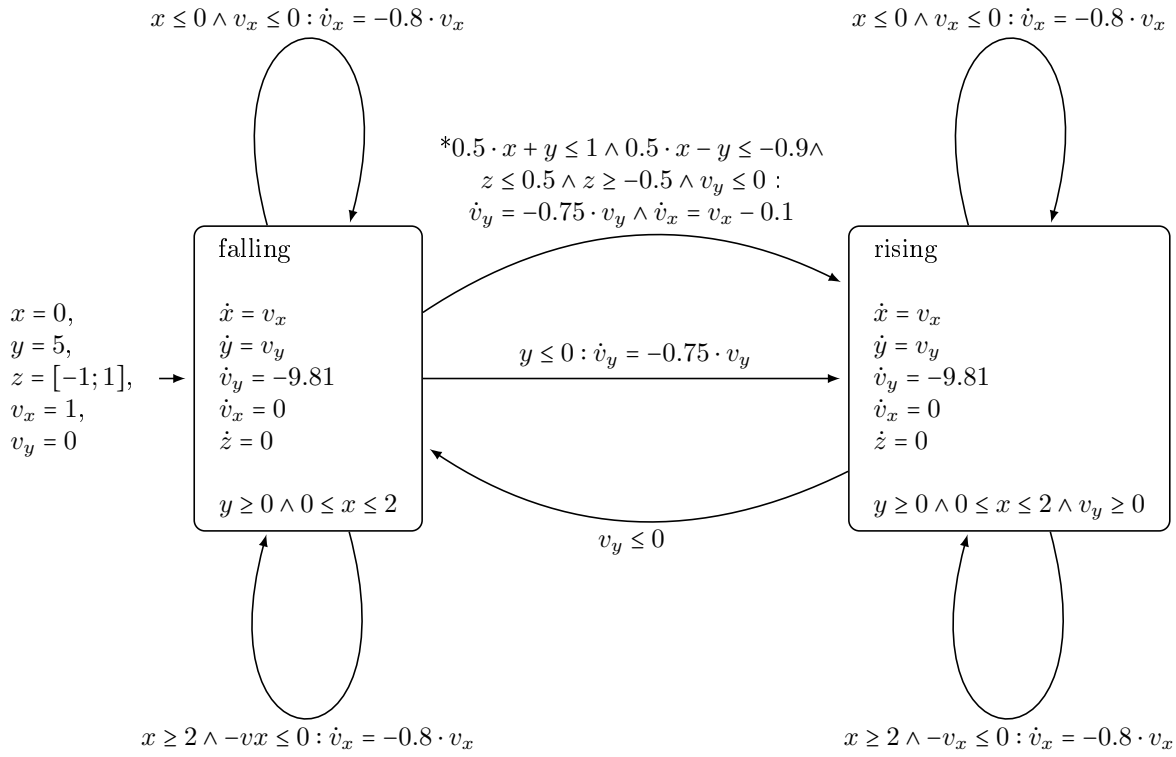

Figure 5.3: Bouncing Ball Automaton .
* marks a transitions as urgent

5.2 Bouncing Ball

The second benchmark is an adapted version of the bouncing ball. The system consists of a ball, that bounces up and down and left and right. It starts at a given height and a given velocity in the horizontal direction. While bouncing the ball is bounded in its movement by two walls left and right and the floor. In addition, there is a tilted beam at some height, that the ball also can hit.
In this scenario, the automaton modeling this system, should ensure, that the ball cannot move through the beam. In order to realize that, the transition, that controls the bouncing onto the beam, is urgent. The automaton modeling this system is shown in Figure 5.3. In this automaton the y-variable controls the height of the ball and the x-variable the left to right movement. The v-variables control the corresponding speed in each direction. In addition, there is a z-variable. This variable stands for the position concerning the back and forth movement, but as we can see, there is no movement in this direction, but a constant position value between -1 and 1. This interval in the initial configuration is the reason, why the set difference is needed in this scenario. The beam, which causes the urgency, does not extend over the complete z interval, but only over a part of it. Therefore, at one point in the reachability analysis not the whole state set will satisfy the urgent guard, but part of it does.

Table 5.2: Comparison Urgency vs. No-Urgency, Time in Seconds

Number of Jumps	1	2	3	4	5
Non-Urgent Automaton	11.6	23.1	52.0	206	669
Urgent Automaton	13.7	31.4	67.2	216	983

In a first test scenario the unsafe state is represented by the position $y = 0, x = 1, z = 0$ and the velocity $v_y \leq 0$ in the falling location. This corresponds to the position central under the beam, which should not be reached. In this scenario we see, that the urgent transition indeed has to be urgent to ensure safety. Without the urgency it shows, that during the reachability analysis a bad state is reachable, while the system is safe with urgency. The reason for that again is quite obvious. Without the transition, that controls the bouncing on the beam being urgent, the transition can be taken, but it does not have to. So the ball can bounce on the beam and start rising again, but can also just fall through the beam and therefore reach the unsafe state.
In a second scenario we consider the system without any unsafe state, to compare the runtime of the system with and without urgency. These times are shown in Table 5.2. In this scenario we can already see a huge difference in the runtime with a small number of jumps, that are taken. The reason for that is the set difference operation, that has to be used every time the urgent transition is satisfied. In addition, especially the last column of Table 5.2 shows, that the more jumps can be taken and correspondingly the more often the computation has to be split off when using urgency, the additional time, that is needed for the set difference, increases very much. In this case the CEGAR approach, shown in [Ebe21], might be interesting. In both cases the runtimes are already large for a small number of jumps. Here the reason is, that polytopes require much more computational effort, than boxes. Although the polytope is already reduced every time a new flowpipe construction starts, there are still many half-spaces, that are used to represent every single one. Additionally, this automaton consists of five variables, which requires even more computational effort.

Chapter 6

Conclusion

The goal of this work wa to implement a set difference operation for the state set representations boxes and polytopes and integrate urgency in the reachability analysis of hybrid systems using this methods.
The set differnce methods have been introduced in Chapter 3 and are implemented in the C++ library HyPro. Additionally, Chapter 4 shows, how to adapt the reachability analysis of hybrid systems, in order to intregerate urgent transitions. These implementations have also been tested in Chapter 5 with the result, that they work correctly, but may have to be improve, in order to improve the runtime. Generally we can say, that the integration of urgency in hybrid systems allows more scenarios to be modeled, but also brings a lot more computational effort.

6.1 Future work

There are some interesting points related to the reachability analysis of hybrid systems using urgency.

- Heuristics in the set difference computation: as shown in Chapter 3 the result of the set difference of two sets, for both boxes and polytopes, is not unique. It depends on the order, in which the dimensions for boxes or the half-spaces for polytopes, are considered by computing the set differnce. In the context of the reachability analysis of hybrid systems it might be interesting to design heuristics for choosing the order. Such heuristics could for example depend on the flow of the system.

- Aggregation and clustering in the reachability anaylsis of hybrid systems using urgency: in this work we did not use aggreration or clustering for the reason shown in Chapter 4. Nevertheless these concepts could speed the computation up for some hybrid system by reducing the number of *ReachTree*-Nodes, that are computet. It would be interesting to implement different approaches for these concepts. There are possibilities to apply aggreration or clustering either before or after computing the set difference.

- Shadow of urgent transitions: as shown inChapter 4 the guard of an urgent transition casts a shadow, which has to be subtracted from the remaining state set. This problem could maybe be approached by constructing a half-space

from the flow of the system or even apply time evolution to the guard, but these approaches also lead to new challenges.

- General hybrid automata: in this work, we hab a look at linear system, but urgency gets really useful, when there is for example also non linear flow. Non linear flow could lead to cases, where the guard of a transition is just touched at one point, which makes the whole scenario much more interesting.

Bibliography

[Bao05] Mato Baotic. *Optimal control of piecewise affine systems: A multi-parametric approach*. PhD thesis, ETH Zurich, 2005.

[BMDP02] Alberto Bemporad, Manfred Morari, Vivek Dua, and Efstratios N Pistikopoulos. The explicit linear quadratic regulator for constrained systems. *Automatica*, 38(1):3–20, 2002.

[Ebe21] Tristan Ebert. Cegar approach for handling urgency in hybrid systems. Master's thesis, RWTH Aachen Universität, 2021.

[Grü13] Branko Grünbaum. *Convex polytopes*, volume 221. Springer Science & Business Media, 2013.

[GV05] B. Gebremichael and F. Vaandrager. Specifying urgency in timed i/o automata. In *Third IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 64–73, 2005.

[Her10] Holger Hermanns. Prohver casestudies, 2010.

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

[NOSY92] Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. An approach to the description and analysis of hybrid systems. In *Hybrid Systems*, pages 149–178. Springer, 1992.

[SÁC+15] Stefan Schupp, Erika Ábrahám, Xin Chen, Ibtissem Ben Makhlouf, Goran Frehse, Sriram Sankaranarayanan, and Stefan Kowalewski. Current challenges in the verification of hybrid systems. In *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, pages 8–24. Springer, 2015.

[SAMK17] Stefan Schupp, Erika Abraham, Ibtissem Ben Makhlouf, and Stefan Kowalewski. Hypro: A c++ library of state set representations for hybrid systems reachability analysis. In *NASA Formal Methods Symposium*, pages 288–294. Springer, 2017.

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

[SFÁ19] Stefan Schupp, Goran Frehse, and Erika Ábrahám. State set representations and their usage in the reachability analysis of hybrid systems. Technical report, Fachgruppe Informatik, RWTH Aachen Universität, 2019.

[SM14] Goran Frehse Stefano Minopoli. Non-convex invariants and urgency conditions on linear hybrid automata. 2014.

[vBRSR07] Dirk A van Beek, Michel A Reniers, Ramon RH Schiffelers, and Jacobus E Rooda. Foundations of a compositional interchange format for hybrid systems. In *International Workshop on Hybrid Systems: Computation and Control*, pages 587–600. Springer, 2007.

[ZSR+10] Lijun Zhang, Zhikun She, Stefan Ratschan, Holger Hermanns, and Ernst Moritz Hahn. Safety verification for probabilistic hybrid systems. In *CAV*, volume 6174 of *LNCS*, pages 196–211. Springer, 2010.

Appendix

Algorithm 10 ComputeTimeSuccessors-function using urgent transitions 1/3

```
template <typename Representation>
template <typename OutputIt>
REACHABILITY_RESULT LTISetMinusWorker<Representation>::computeTimeSuccessors(const Representation&
  initialSet, Location<Number> const* loc, OutputIt out) const {
      //constructing the first segment
      Representation firstSegment = constructFirstSegment(initialSet, loc->getLinearFlow(),
               mTrafoCache.transformationMatrix(loc, mSettings.timeStep), mSettings.timeStep);
      //vector for result of setminus
      std::vector<Representation> result;
      //vector of urgent transitions in current location
      std::vector<const Transition<Number>*> urgent_trans;
      //vector of flowpipe segments
      std::vector<Representation> segments;
      //variables used to keep track of the computation
      std::vector<Representation> tmpsegments;
      int lower = 0;
      int tmplower = 0;
      int urgentlower = 0;
      int tmpurgentlower = 0;
      int upper = 1;
      int tmpupper = 1;
      int urgentupper = 1;
      int tmpurgentupper = 1;
      //intersect with invariant
      auto [containment, segment] = intersect(firstSegment, loc->getInvariant());
      //If the first segment did not fulfill the invariant of the location,
        the jump here should not have been made
      assert(containment != CONTAINMENT::NO);
      segments.push_back(segment);
      //getting all urgent actions
      for (const auto& transition : loc->getTransitions()) {
            if (transition.get()->isUrgent()) {
                  urgent_trans.push_back(transition.get());
            }
      }
```

Algorithm 11 ComputeTimeSuccessors-function using urgent transitions 2/3

```
//check if urgent jump is enabled
for (unsigned long int i = 0; i < urgent_trans.size(); i++) {
	for (int ut = tmplower; ut < tmpupper; ut++) {
		segment = segments[ut];
		//check if transition is enabled
		auto [containment, testsegment] = intersect(segment, urgent_trans.at(i)->getGuard());
		//if guard is satisfied, perform setminus
		if (!testsegment.empty()) {
			Representation guard(urgent_trans.at(i)->getGuard().getMatrix(),
					urgent_trans.at(i)->getGuard().getVector());
			result = segment.setMinus2(guard);
		}
		// insert segments into flowpipe
		if (result.size() > 0) {
			for (unsigned long int i = 0; i < result.size(); i++) {
				if (!(result.at(i).empty())) {
					if (i == 0 && urgentlower == tmplower) {
						urgentlower = tmpupper;
					}
					urgentupper++;
					segments.push_back(result.at(i));
				}
			}
		}
		else {
			if (!segment.empty()) {
				if (urgentlower == tmplower) {
					urgentlower = tmpupper;
				}
				segments.push_back(segment);
				urgentupper++;
			}
		}
	}
	tmplower = urgentlower;
	tmpupper = urgentupper;
}
lower = tmplower;
upper = tmpupper;
for (int i = tmplower; i < tmpupper; i++) {
	//add to flowpipe of node
	*out = segments[i];
	++out;
	// intersect with badstates
	std::tie(containment, std::ignore) = ltiIntersectBadStates(segments[i], loc, mHybridAutomaton);
	if (containment != CONTAINMENT::NO) {
		// Todo: memorize the intersecting state set and keep state.
		return REACHABILITY_RESULT::UNKNOWN;
	}
}
urgentlower = 0;
urgentupper = 1;
tmpurgentlower = urgentlower;
tmpurgentupper = urgentupper;
//while timebound not reached
for (size_t segmentCount = 1; segmentCount < mNumSegments; ++segmentCount) {
	//for all segments of previous timeevolutionstep
	for (unsigned long int anz = lower; anz < upper; anz++) {
		//let time elapse
		segment = applyTimeEvolution(segments[anz],
				mTrafoCache.transformationMatrix(loc, mSettings.timeStep));
		result.resize(0);
		//check invariant
		std::tie(containment, segment) = intersect(segment, loc->getInvariant());
		if (containment == CONTAINMENT::NO) {
			return REACHABILITY_RESULT::SAFE;
		}
```

Algorithm 12 ComputeTimeSuccessors-function using urgent transitions 3/3

```
                    tmpsegments.resize(0);
                    tmpsegments.push_back(segment);
                    urgentlower = 0;
                    urgentupper = 1;
                    tmpurgentlower = urgentlower;
                    tmpurgentupper = urgentupper;
                    //check if urgent jump is enabled
                    //for all urgent transitions
                    for (unsigned long int i = 0; i < urgent_trans.size(); i++) {
                          //for all parts of the current segment
                          for (int ut = urgentlower; ut < urgentupper; ut++) {
                                segment = tmpsegments[ut];
                                if (!segment.empty()) {
                                      auto [containment, testsegment] = intersect(segment,
                                                                                  urgent_trans.at(i)->getGuard());
                                }
                                else {
                                      auto containment = CONTAINMENT::NO;
                                }
                                //if guard is enabled perform setminus
                                if (containment != CONTAINMENT::NO) {
                                      Representation guard(urgent_trans.at(i)->getGuard().getMatrix(),
                                                           urgent_trans.at(i)->getGuard().getVector());
                                      result = segment.setMinus2(guard);
                                      //insert segment
                                      if (result.size() > 0) {
                                            for (unsigned long int i = 0; i < result.size(); i++) {
                                                  if (tmpurgentlower == urgentlower) {
                                                        tmpurgentlower = urgentupper;
                                                  }
                                                  tmpurgentupper++;
                                                  tmpsegments.push_back(result.at(i));
                                            }
                                      }
                                      else {
                                            if (tmpurgentlower == urgentlower) {
                                                  tmpurgentlower = urgentupper;
                                            }
                                            tmpurgentupper++;
                                            tmpsegments.push_back(segment);
                                      }
                                }
                          }
                          urgentlower = tmpurgentlower;
                          urgentupper = tmpurgentupper;
                    }
                    //insert segments to flowpipe of node, if its not empty
                    for (int i = urgentlower; i < urgentupper; i++) {
                          auto seg = tmpsegments[i];
                          if (!seg.empty()) {
                                segments.push_back(seg);
                                *out = seg;
                                ++out;
                                if (tmplower == lower) {
                                      tmplower = upper;
                                }
                                tmpupper++;
                                // intersect with badstates
                                std::tie(containment, std::ignore) = ltiIntersectBadStates(seg,
                                                                                           loc, mHybridAutomaton);
                                if (containment != CONTAINMENT::NO) {
                                      // Todo: memorize the intersecting state set and keep state.
                                      return REACHABILITY_RESULT::UNKNOWN;
                                }
                          }
                    }
              }
              lower = tmplower;
              upper = tmpupper;
        }
        return REACHABILITY_RESULT::SAFE;
}
```