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

MASTER OF SCIENCE THESIS

ADAPTIVE DYNAMIC REACHABILITY ANALYSIS FOR LINEAR HYBRID AUTOMATA
ADAPTIVE, DYNAMISCHE ERREICHBARKEITSANALYSE FÜR LINEARE HYBRIDE AUTOMATEN

Dustin Hütter

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

Additional Advisor:
M. Sc. Stefan Schupp

Aachen, 29.09.2016

Abstract

Hybrid systems exhibit discrete and continuous behavior. Occurring e.g. in cyber-physical systems, verifying whether such systems do not reach certain undesired states is highly important because a lot of these systems are safety-critical.
There already exists a set of algorithms computing over-approximations of the set of reachable states of hybrid systems. This thesis extends a common static approach for the reachability analysis to a dynamic one. The presented dynamic approach enables us to refine single paths of a given model where the set of reachable states invalidated a given safety criterion while avoiding the re-computation of the whole model. Additionally, we exploit previous results of our analysis and do not have to restart it for the whole system.
Our evaluation indicates that this procedure can make reachability analysis more feasible in terms of runtime and flexibility.

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

Acknowledgements

Firstly, I would like to thank my advisor Stefan Schupp for his steady support of this thesis project and the fruitful discussions that we had. Additionally, I want to thank the whole Theory of Hybrid Systems group and especially Prof. Dr. Erika Ábrahám for supporting me and giving me the chance to write this thesis in an interesting field of research. My last thanks goes to Prof. Dr. Jürgen Giesl for his willingness to be the second supervisor of this thesis.

Contents

Chapter 1

Introduction

In order to ensure that certain systems occurring in science and industry have desired properties, the research area of verification has evolved. On an abstract level, its aim is to build an abstraction of the system and to apply formal methods on this system to judge whether the abstraction satisfies the properties or not. This thesis focuses in particular on recognizing whether the modeled systems do not reach undesired states. These properties are also called *safety properties*.
While this problem is well-understood for systems exhibiting only discrete or only continuous behavior, the verification of hybrid systems, featuring both discrete and continuous behavior, is a rather emerging field of research. Exemplifying such a system, consider a controller for rods of a reactor. We may have discrete states modeling either that we cool them down or heat them up. The physical change of temperature in contrary is continuous. We will introduce the notion of *hybrid automata* which we will use to build an abstraction of such a system. Our aim is then to judge whether a (linear) hybrid automaton avoids a certain set of bad states. For the rod reactor example such a bad state could be that the temperature of a rod exceeds a certain value.

Firstly, we will introduce the necessary theoretical foundations of hybrid automata and have a look at an existing approach for the reachability analysis of linear hybrid automata. As this problem is undecidable [HKPV95], the latter approach restricts itself to over-approximating the precise reachable set. In case the over-approximation has an empty intersection with the bad states, we can deduce that the actual reachable set also does. Otherwise we can refine the reachability analysis. If the over-approximation still intersects the bad states, we can not deduce that the bad states are not reachable because it may be that only the set difference of the over-approximation and the precise reachable set intersects the bad states.
The main contribution of this thesis is an extension of the previously mentioned approach to a dynamic one. The algorithm that we build on is static in the way that we can initially specify a set of parameters, proceed with the analysis and get the answer whether the over-approximation intersects the bad states or not. In case the intersection is non-empty, we can restart the analysis with a new parameter setting. Using a more precise parameter setting has a crucial effect on the quality of the over-approximation. The theory part formalizes the notions of the quality of parameter settings and over-approximations. As a matter of fact, the static approach can be optimized in several ways. Firstly, with the dynamic approach we are able to reuse

the results of previous runs. Secondly, it may be beneficial to detect the part that was responsible for the non-empty intersection with the bad states and to only repeat the reachability algorithm for this part. In Chapter 3 we will present the idea and implementation issues of our dynamic reachability algorithm. Thereafter, we will evaluate it and examine its advantages and disadvantages. In the last chapter we will conclude this thesis and explore possible optimizations for the future.

Chapter 2

Theoretical Background

In this chapter we will introduce the necessary theoretical foundations that the main notions of this thesis base on. Therefore, we will first introduce the model that we aim to model-check in terms of reachability. Then, we will present several technical details leading to a common algorithm for reachability analysis.

2.1 Hybrid Automata

As we have already stated, we aim to have a formal model for systems featuring both discrete and continuous behavior i.e. hybrid systems. Hybrid automata provide us such a model. They enable us to build an abstraction of the modeled hybrid systems capturing its main characteristics. We will first introduce *general hybrid automata* and afterwards a more restrictive model, *linear hybrid automata*, that we will use for further considerations. The different classes of hybrid automata are distinguished by their expressiveness and decidability of common problems involving these automata [HKPV95].

2.1.1 General Hybrid Automata

In the following, we introduce the model of *general hybrid automata*. Subsequently, we will simply denote them by *hybrid automata* [ACHH93]. A hybrid automaton H is a 7-tuple of the form $H = (Loc, Var, Lab, Edge, Act, Inv, Init)$ where:

- Loc is a finite set of *locations*. They correspond to the states in finite automata or Kripke structures.
- Var is a finite set of real-valued variables. They facilitate the continuous part of the model by allowing to let them evolve in each location. In the following, let $d = \mid Var \mid$.
- Lab is a finite set of synchronization labels including a stutter label τ. These labels allow to describe the action triggering the transition. In case there is no explicit synchronization label, it is implicitly assumed to be τ.
- $Edge$ is a finite set of edges with $Edge \subseteq Loc \times Lab \times 2^{\mathbb{R}^d \times \mathbb{R}^d} \times Loc$. The notion of a transition is given by $t = (l,a,\mu,l') \in Edge$ where μ is a so-called *reset map*.

The first component of $\mu \in 2^{\mathbb{R}^d \times \mathbb{R}^d}$ contains the valid variable valuations with which a transition can be taken. The second component describes the variable valuations after the transition was taken. A transition t, triggered by the label a, can be taken from state $(l,v) \in Loc \times \mathbb{R}^d$ to $(l',v') \in Loc \times \mathbb{R}^d$ iff $v' \in Inv(l')$ and $(v,v') \in \mu$ i.e. v satisfies the ***guard*** of t and assigns the variables according to v'. In case the guard of a transition is satisfied, it is said to be *enabled.*

- Act is a function assigning a set of activities $f : \mathbb{R}^+ \to \mathbb{R}^d$ to each location. For each point in time, f models the dynamic of the variables in a certain location by a differential equation.

- Inv is a function assigning an *invariant* $Inv(l) \subseteq \mathbb{R}^d$ to each location $l \in Loc$ i.e. determining a set of valid valuations for each location.

- $Init$ is a set of initial states.

Note that due to the fact that we have locations and evolving variables, unlike as for Kripke structures, states are not solely given by the location but by a pair $(l,v) \in Loc \times \mathbb{R}^d$ of the current location and variable valuation.

In Figure 2.1 we see an exemplary hybrid automaton H with the formal description:

- $Loc = \{l_0,l_1\}$,

- $Var = \{x_0,x_1\}$,

- $Lab = \{a,b,\tau\}$,

- $Edge = \{(l_0,a,\{(v,v') \in \mathbb{R}^d \times \mathbb{R}^d : v(x_1) \geq 5 \wedge v'(x_0) = 0 \wedge v(x_1) = v'(x_1)\},l_1),(l_1,b,\{(v,v') \in \mathbb{R}^d \times \mathbb{R}^d : v(x_1) \leq 5 \wedge v'(x_0) = 0 \wedge v(x_1) = v'(x_1)\},l_0)\}$,

- $Act(l_0) = \{f : \mathbb{R}^+ \to \mathbb{R}^d : \exists c_1,c_2 \in \mathbb{R}\ \forall t \in \mathbb{R}^+ (f(t)(x_0) = t + c_1 \wedge f(t)(x_1) = 2t + c_2)\}$,
  $Act(l_1) = \{f : \mathbb{R}^+ \to \mathbb{R}^d : \exists c_1,c_2 \in \mathbb{R}\ \forall t \in \mathbb{R}^+ (f(t)(x_0) = t + c_1 \wedge f(t)(x_1) = -2t + c_2)\}$

- $Inv(l_0) = \{v \in \mathbb{R}^d : v(x_0) \leq 10 \wedge v(x_1) \leq 10\}$, $Inv(l_1) = \{v \in \mathbb{R}^d : v(x_0) \leq 10 \wedge v(x_1) \geq 0\}$,

- $Init = \{(l_0,(0,0))\}$.

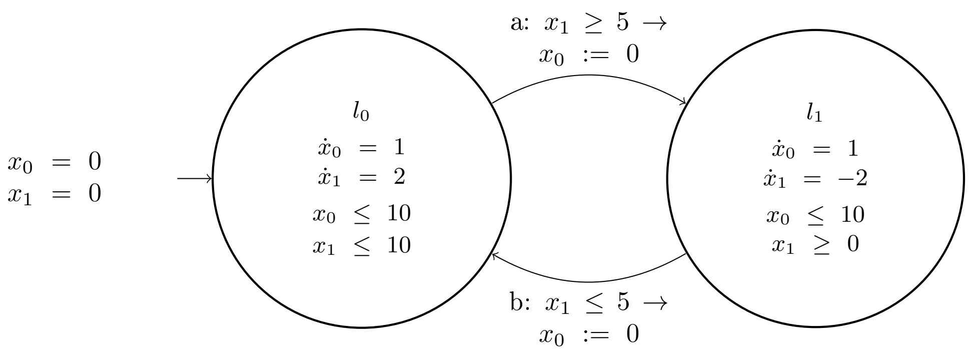


Figure 2.1: Graphical representation of the exemplary hybrid automaton.

H consists out of the two locations l_0 and l_1 and the variables x_0 and x_1. Its single initial state is $(l_0,(0,0))$. In l_0, x_0 and x_1 are forced to be less or equal than 10 and in l_1 x_0 is also forced to be less or equal than 10 and x_1 to be greater or equal than 0. Furthermore, we have one discrete transition from l_0 to l_1 which is triggered by the label a and resets x_0 to 0. The other transition is triggered by the label b and enabled when x_1 is less or equal than 5 and also resets x_0 to 0.

2.1.2 Formal Semantics

In order to have a proper basis to define the reachability problem that we aim to solve, we will introduce the formal one-step semantics of hybrid automata. As we have seen, the state of a hybrid automaton is a pair of a location and a variable valuation. Hence, a state change may either take place by changing the location and resetting certain variable valuations i.e. by taking a discrete transition or by just changing the variable valuations i.e. by letting the variables evolve according to the dynamics in the current location. The former is given by:

$$\frac{(l,a,\mu,l') \in Edge \quad (v,v') \in \mu \quad v' \in Inv(l')}{(l,v) \xrightarrow{a} (l',v')} \; Rule_{Discrete}$$

A discrete state change $(l,v) \xrightarrow{a} (l',v')$ accordingly obeys the premises that a transition $(l,a,\mu,l') \in Edge$ exists such that its guard is satisfied and the new variable valuation v' satisfies the invariant of the location l' the discrete transitions ends in.
The second way of a state change is defined by:

$$\frac{f \in Act(l) \quad f(0) = v \quad f(t) = v' \quad t \geq 0 \quad \forall 0 \leq t' \leq t . f(t') \in Inv(l)}{(l,v) \xrightarrow{t} (l,v')} \; Rule_{Time}$$

A continuous state change $(l,v) \xrightarrow{t} (l,v')$ obeys the premises that for the activity f in the location l, the initial valuation v changes to $v' = f(t)$ and for every point in time that is in between, including 0 and t, the invariant of l is satisfied.

We say that a state (l,v) is reachable in a hybrid automaton H if there is a sequence of transitions, either discrete or continuous, $(l_0,v_0) \rightarrow ... \rightarrow (l,v)$ with l_0 being an initial location and v_0 a corresponding initial valuation of H. In particular, given a set of bad states $B = \{(l_{b_1},v_{b_1}),...,(l_{b_n},v_{b_n})\} \in Loc \times \mathbb{R}^d$, we are interested in determining whether a state in B is reachable from (l_0,v_0). Due to the undecidability of the reachability problem for hybrid automata, we will restrict ourselves to only compute over-approximations of the reachable set. Therefore, if the over-approximation does not intersect the reachable set, we can deduce that the reachable set also does not. Furthermore, we will restrict the dynamics of the locations leading us to *linear hybrid automata.*

2.1.3 Linear Hybrid Automata

Linear hybrid automata are a special type of the hybrid automata that we have already introduced. An autonomous linear hybrid automaton is a hybrid automaton with the additional restriction that the activities of the variables in each location are given by *linear ordinary differential equations* (linear ODEs) exhibiting the following form [LG09]:

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

respectively

$$\dot{x}(t) = Ax(t) + u(t) \tag{2.2}$$

for non-autonomous linear hybrid automata. Here, $A \in \mathbb{R}^{d \times d}$ is a matrix of coefficients for the d variables of the corresponding linear hybrid automaton. For the non-autonomous case, the dynamics additionally possess a function $u(t)$ with which we can model external input. However, in this thesis we will only consider the autonomous case. Determining solutions of linear ODEs enables us to compute the development of variables' values in a single location which obviously is important in reachability analysis. Solutions of linear ODEs as in Equation 2.1 have the following form [LG09]:

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

where x_0 is the initial value of x and e^{At} denotes the so called *matrix exponential* given by [Leo96]:

$$e^{At} = \sum_{i=0}^{\infty} \frac{A^i t^i}{i!}.$$

Hence, in comparison to general differential equations, solutions of linear ODEs can be computed with moderate effort. Due to this fact, several approaches for the reachability analysis of hybrid automata restrict themselves to linear hybrid automata just as this thesis does.

2.2 General Reachability Analysis

As this thesis is devoted to the reachability analysis of linear hybrid automata, in this section we will make a first step towards the reachability algorithm that we will later use and extend. Algorithm 1 from [Á15] gives a rough sketch of how we can employ such an analysis. Starting from the initial states $Init_H$ of a hybrid automaton H, we iteratively compute an over-approximation of the reachable set. Technically, in one iteration the call *Reach*(..) in Line 5 computes the flowpipes for all states that resulted from taking a discrete transition in the last iteration respectively computes the flowpipes for the initial states in the first iteration. After computing the flowpipe, *Reach*(..) checks which guards of outgoing transitions are satisfied and applies the reset of the variable valuations to the satisfying sets for these transitions. The variable R_{new} stores these sets provided by applying the reset of the variable valuations without the state sets that have been computed before i.e. R_{new} is assigned with $\text{Reach}(R_{new})\backslash R_H$. Initially, R_{new} is assigned with $Init_H$ in Line 1. R_H stores all states that were determined during the reachability algorithm. In case R_{new} is empty, our analysis terminates with returning R_H. Otherwise, there exists a state that can still be evolved and the while loop continues. Additionally, to ensure termination the maximum number of discrete jumps on each path and a *time horizon* T i.e. the time that we stay at most in one location can be fixed.

Algorithm 1 General Reachability Analysis

```
Input: A hybrid automaton $H$ with initial states $Init_H$
Output: Set $R_H$ of reachable states
1: $R_{new} := Init_H$
2: $R_H := \emptyset$
3: while $R_{new} \neq \emptyset$ do
4:     $R_H := R_H \cup R_{new}$
5:     $R_{new} := \text{Reach}(R_{new}) \setminus R_H$
6: end while
7: return $R_H$
```

As a first approach for the reachability analysis of hybrid automata, this algorithm is suitable but it clearly abstracts from a lot of technical details e.g. how exactly *Reach*(...) proceeds. In the following we will introduce these technical details that we need to give a proper implementation.

2.3 Representations

So far, we abstracted from the way in which we represent the reachable set respectively an over-approximation of it. As a matter of fact, the choice of the representation is crucial regarding the accuracy of the approximation and the efficiency of the analysis procedure. In the following, we will introduce boxes, polytopes, zonotopes and support functions. The interested reader finds a more opulent collection of representations e.g. in [LG09].

2.3.1 Boxes

A simple representation is given by boxes. They are defined as follows:

Definition 2.3.1 (Box [LG09])**.**

A set B is a box iff it can be expressed as a product of intervals.

$$B = [\underline{x_1}, \overline{x_1}] \times ... \times [\underline{x_d}, \overline{x_d}]$$

B is the set of points x whose i^{th} coordinate, x_i, lies between $\underline{x_1}$ and $\overline{x_1}$.

Intuitively, boxes approximate a set by a hyper-rectangle. Their main advantage is that they can be stored efficiently only using $2d$ parameters in d dimensions and a lot of frequently used operations in reachability analysis can be performed quite efficiently with boxes. One of their downsides is, shown in Figure 2.2, that the introduced over-approximation is in general large in comparison to other state set representations.

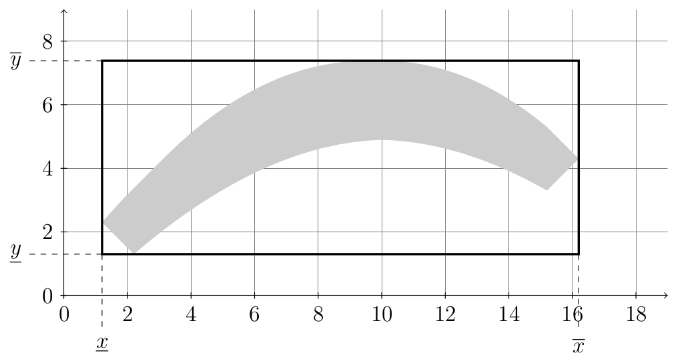


Figure 2.2: A box over-approximating another set [LG09].

2.3.2 Polytope

A more accurate representation is given by means of polytopes. Therefore, we first define the notion of a *half-space*:

Definition 2.3.2 (Halfspace [Zie95])**.**

A d-dimensional half-space h is a set of points with $h = \{x \in \mathbb{R}^d : c^T x \leq z\}$, with $c \in \mathbb{R}^d$ being the normal vector of h and $z \in \mathbb{R}$ being an offset parameter.

Definition 2.3.3 (Polytope [Zie95])**.**

A polytope P is the bounded intersection of a finite set H of half-spaces:

$$P = \bigcap_{h \in H} h$$

One way of representing polytopes is to store the half-spaces defining the polytope. This form of storing polytopes is called ***H-Polytopes***. Another way is to store a finite set V of vertices and to define a polytope as the convex hull of V. The notion of a convex hull is explained in Section 2.4. Both ways of representing polytopes have their advantages and disadvantages regarding the execution of certain frequently performed operations. Furthermore, they are equivalent in terms of expressivity. Hence, one can combine both representations but as a matter of fact transforming one representation into the other goes along with exponential cost in the worst-case [Zie95]. Figure 2.3 illustrates a set being over-approximated by a polytope. We can observe that the polytope yields a finer approximation as boxes did. However, this benefit goes along with higher computational costs.

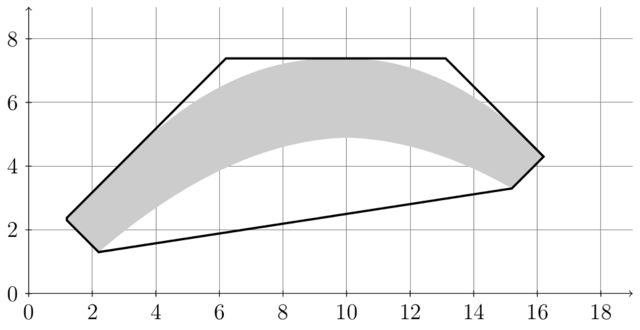


Figure 2.3: A polytope over-approximating another set [LG09].

2.3.3 Zonotope

A further representation is given by zonotopes. They may be defined as follows [LG09]:

Definition 2.3.4 (Zonotope)**.**

A zonotope Z may be defined as follows:

$$Z = \{x \in \mathbb{R}^d : x = c + \sum_{i=1}^{d} \alpha_i g_i\},$$

where the $g_i \in \mathbb{R}^d$ are the so-called generators of a generator set $G = \{g_1,...,g_d\}$, the $\alpha_i \in [-1,1]$ for $i \in \{1,...,d\}$ are scalars and $c \in \mathbb{R}^d$ is the center of Z.

Zonotopes are also a quite common form of representing reachable sets. However, they have a center of symmetry and therefore generally do not allow as much precision in approximating a set as for example polytopes do. On the other side, zonotopes exhibit constant runtime complexity e.g. for linear transformations which is an important operation for reachability analysis as we will see. Figure 2.4 gives an illustration of a zonotope.

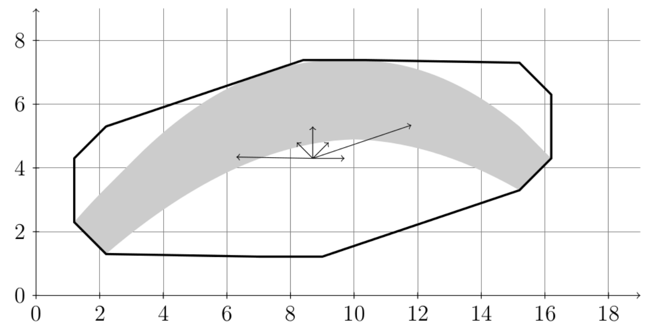


Figure 2.4: A zonotope over-approximating another set [LG09].

2.3.4 Support Functions

Here, we will consider the representation given by so-called support functions. They differ from the representations seen so far by the fact that they are not given by a collection of parameters but by a symbolic representation. Their definition is as follows:

Definition 2.3.5 (Support function [LG09]).

The support function of a set S, denoted ρ_S is defined by:

$$\begin{aligned} \rho_S: \ & \mathbb{R}^d \to \mathbb{R} \cup \{\infty, -\infty\} \\ & l \mapsto \sup_{x \in S} x \cdot l \end{aligned}$$

A point x of S such that $x \cdot l = \rho_S(l)$ is called a support vector of S in direction l.

The underlying concept of support functions is that any direction i.e. a vector $x \in \mathbb{R}^d$ is mapped to a plane fully containing the set to be considered and crossing it in exactly one point. Figure 2.5 visualizes this for one vector l. Thereby, we can approximate a given set by the support functions of several directions. The more directions that are taken into account, the more precise the approximation gets.

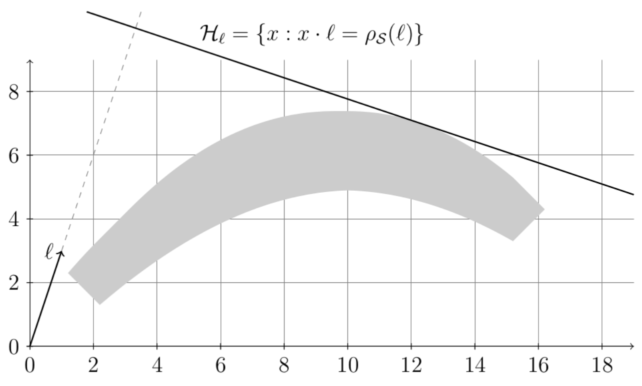

$\mathcal{H}_\ell = \{x : x \cdot \ell = \rho_\mathcal{S}(\ell)\}$
ℓ
0
2
4
6
8
0
2
4
6
8
10
12
14
16
18

Figure 2.5: A set with a supporting hyper-plane in direction l [LG09].

2.4 Operations

In addition to the way in which we represent the over-approximation of the reachable states, we also need to be able to perform certain operations on the calculated sets during the reachability computation. In the following, we will introduce the most important ones.

Two fundamental operations are set union, denoted by $\cup$, and set intersection, denoted by $\cap$, known from classic set theory and defined as $A \cup B := \{x | x \in A \vee x \in B\}$ respectively $A \cap B := \{x | x \in A \wedge x \in B\}$ for two sets A and B. The intersection e.g. is used to determine which part of a set satisfies a guard of some transition. For the union of two sets, we need to add the ingredient of building the *convex hull* of this union for our approach. This is required as the over-approximations that we use are obligated to be convex which the union of two convex sets not necessarily is. We denote the convex hull of a set A by $CH(A)$. It is defined as the smallest convex set containing A. This operation is illustrated in Figure 2.6.

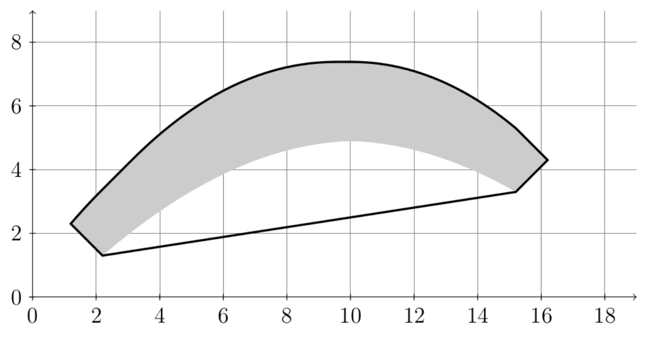


Figure 2.6: A set (shaded gray) and its convex hull [LG09].

Another important operation is the *Minkowski sum*, denoted by $\oplus$. For two sets A and B it is defined as $A \oplus B := \{a + b | a \in A \wedge b \in B\}$. This operation is e.g.

needed for the initialization step of the reachability algorithm when computing the first over-approximation. Figure 2.7 shows an exemplification of the Minkowski sum:

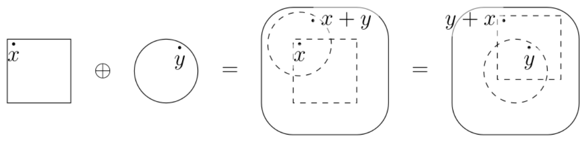


Figure 2.7: Minkowski sum for two sets [LG09].

In the following section, we will see how these operations are used in the reachability algorithm. For a more detailed description of the presented operations, in particular how well each of these can be performed using different representations, the interested reader may consult [LG09].

2.5 Flowpipe-based Reachability Analysis

We are now able to give a refined version of Algorithm 1. As there are two ways for a state set to evolve, namely by changing variable valuations in a single location l according to the dynamic of l and by taking discrete transitions, we will consider both and finally give a pseudocode version of a flowpipe-based reachability analysis [CK98].

2.5.1 Reachability by Variable Evolution in Single Location

One way of transitioning from one state of a hybrid automaton to another one is to let the variables evolve in one location l according to the linear ODEs of the variables in l. As we have seen, a solution of a linear ODE is given by Equation 2.1.3 :

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

Hence, the reachable variables' assignments, say $R_\delta(S)$, in a single location l at time δ starting from an initial set S, are given by:

$$R_\delta(S) = e^{\delta A}S$$

We use this to compute the over-approximations $\Omega_0, ..., \Omega_T$ of the actual reachable sets. Here, T is the *time horizon* i.e. the amount of time that we stay in a location at most. Furthermore, an Ω_i with $i \in \{0, ..., T\}$ is the over-approximation of the reachable set for the time interval $[\delta i, \delta(i+1)]$. In addition to that, δ is the *time step* that each Ωi covers being an integer divisor of T. Thus, choosing a smaller δ provides a more accurate over-approximation of the reachable sets but simultaneously increases the number of intervals to be considered. Putting this together, the idea is to compute the sequence of over-approximations by means of a *recurrence relation*:

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

That is, we compute the single sets $\Omega_1, ..., \Omega_T$ consecutively by taking their predecessor regarding the time step into account and let them evolve according to the current locations' dynamic.

Construction of Ω_0

A remaining problem is that we need to construct the initial flowpipe segment Ω_0 in order to be able to use the recurrence relation to determine the following flowpipe segments. We will give a brief sketch on how this construction works. As the problem of determining the precise reachable set for linear hybrid automata is undecidable we compute an over-approximation of it. Say (l, X_0) is an element of the initial state set of a linear hybrid automaton to be considered, then the initial flowpipe segment Ω_0 for this element of the initial state set may be determined according to [LG09]:

$$\Omega_0 = CH(R_0(X_0) \cup R_\delta(X_0)) \oplus B(\alpha_\delta)$$

Hence, we take the convex hull of the union of $R_0(X_0)$ i.e. $e^{0A}X_0 = X_0$ and $R_\delta(X_0)$ both being part of the exact flowpipe. This operation results in Ω_0' of Figure 2.8 in which the dotted lines depict the precise dynamic of the current location. We can see that not all trajectories of it are fully contained in the convex hull that we have computed. In order to obtain an over-approximation, we apply the so-called *bloating* mechanism. Technically, this is done by calculating the Minkowski sum of the previous convex hull and $B(\alpha_\delta)$. $B(\alpha_\delta)$ is a ball with radius α_δ which is an upper bound for the *Hausdorff distance* between the precise flowpipe and the over-approximation. We define the Hausdorff distance for two sets X and Y as [LG09]:

$$d_H(X,Y) = \max\{\sup_{x \in X} \inf_{y \in Y} \|x - y\|, \sup_{x \in X} \inf_{y \in Y} \|x - y\|\}$$

Intuitively, the Hausdorff distance for two sets X and Y is determined by finding the absolute value of the maximum of all shortest path for each $x \in X$ to any $y \in Y$ respectively for each $y \in Y$ to any $x \in X$. Further details for this particular part of the construction can be found e.g. in [Dan00]. After having calculated the aforementioned Minkowski sum, the construction ensures that we obtain a proper superset i.e. an over-approximation, Ω_0 in Figure 2.8, of the exact flowpipe that we can use for further proceeding.

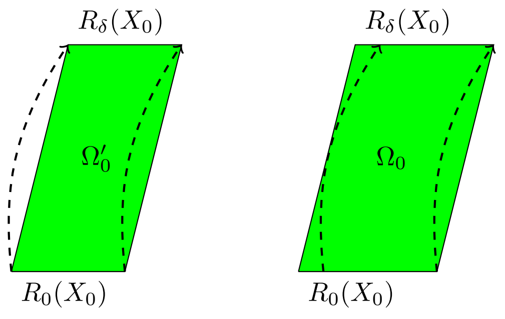


Figure 2.8: Intermediate flowpipe segment Ω_0' not containing all trajectories and the final one Ω_0 being a superset of the precise reachable set.

2.5.2 Reachability by Discrete Transitions

After having presented the mechanism for calculating over-approximations of the exact flowpipe in a single location, in this section we will focus on the second way in which states of a hybrid automaton can evolve, namely by taking discrete transitions. Figure 2.9 illustrates the process of transitioning from a flowpipe in l to a new initial set in another location l' by taking a transition $t = (l,a,\mu,l')$. Basically, we check for all flowpipe segments in l whether the intersection with the states satisfying the respective guard of t is non-empty. For those segments where this is the case, we consider this part and apply the corresponding variable valuation updates of t to it. The usual proceeding is shown in the left part of Figure 2.9. The flowpipe segments Ω_i and Ω_{i+1} both intersect the guard of t. The union of the sets obtained by updating the variable valuations of each Ω_i and Ω_{i+1} is not necessarily convex. The dotted line indicates the part of the union that would break the convexity characteristic. Hence, we compute the convex hull of the union illustrated by Ω_{new} and aggregate the single sets that we obtain by applying the reset map of t to Ω_i and Ω_{i+1}. Another way is to apply non-aggregating transitions as shown in the right part of Figure 2.9. Therefore, we omit the aggregating step and proceed independently with all sets, Ω_{new_1} and Ω_{new_2}, that we obtain by applying the reset map of t to the flowpipe segments of the source location i.e. Ω_i and Ω_{i+1}. On the one hand it provides the advantage of not introducing a further approximation error by not computing the convex hull of the newly obtained sets but on the other hand it dramatically increases the computation effort of the whole reachability analysis because in the following for all new sets flowpipes have to be computed.

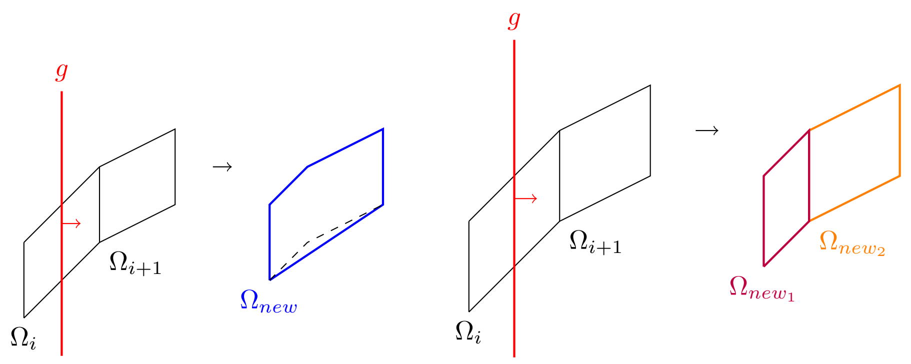


Figure 2.9: An aggregating transition (left part) and a non-aggregating transition (right part).

Putting together what we have elaborated in the last sections, we obtain a pseudocode version, Algorithm 2, of a flowpipe-based reachability analysis [Á15]. At first, R_H is initialized with the initial state set of the input automaton H in Line 1. In R_H the computed reachable states are collected. R_{new} stores exactly those reachable states that have entered R_H in the last step of the computation. The main loop terminates if either no new reachable states have been computed in the last iteration i.e. $R_{new} = \emptyset$ or a termination condition, for example that the maximum number of predefined discrete jumps is reached, is satisfied. In one iteration of the loop, an element of R_{new} is taken into consideration and evolved. This includes to compute the flowpipe for this set and to store the result in the variable R'. Once the flowpipe has been calculated, for all possible transitions whose guards satisfy the flowpipe (segments) the jump successors are determined. The computation of a flowpipe and of jump successors are explained in detail in Section 2.5. The call of computeJumpSuccessor(R') in Line 6 also involves that the newly obtained reachable sets are added to R_{new} respectively R_H.

Algorithm 2 Flowpipe-based Reachability Analysis

```
Input: A linear hybrid automaton $H$ with initial states $Init_H$
Output: Set $R_H$ of reachable states
1: $R_H := Init_H$
2: $R_{new} := R_H$
3: while $R_{new} \neq \emptyset \wedge \neg$termination_cond do
4:     $R_{new} := R_{new} \setminus$ stateSet (for stateSet $\in R_{new}$)
5:     $R' :=$ computeFlowpipe(stateSet)
6:     computeJumpSuccessor($R'$)
7: end while
8: return $R_H$
```

Chapter 3

Dynamic Reachability Analysis

The traditional flowpipe-based approach that we have seen before, is static in the way that we initially fix its parameters like the time step and the representation and keep them for the whole execution of the algorithm. Thereby, some disadvantages come along. For instance, when checking the intersection with a set of bad states, it might be desirable to only compute those reachable sets very precisely that tend to intersect the bad states by taking a small time step or a precise representation and over-approximate the other ones only as fine-grained as necessary. On the one hand it can make reachability analysis more feasible but on the other hand it still allows to determine a precise over-approximation for all parts of a given automaton if desired. In the following, we will introduce our approach for such a dynamic analysis.

3.1 Tree-based Reachability

In order to reach our goal of a dynamic reachability analysis, we will incorporate a tree in the analysis. The nodes of this tree encapsulate the reachable sets for each location respectively the over-approximations of those and enrich them by information about the path of locations that were taken so far, the chosen reachability parameters and several other context information that we will have a close look at in the following. Figure 3.1 illustrates how our tree is constituted. On the first level, which we define to have depth 0, we have an initial node which is basically a dummy node that only references the first actual level of reachable states. Each following level represents the reachable states after having taken one more discrete transition than on the level before. Initially, each node is given a reachable set resulting from the last computation step respectively an initial state set for the nodes of level 1. For this set, we compute the flowpipe i.e. we compute an over-approximation of the reachable variable valuations according to the dynamics of these variables in the current location. E.g. the nodes of depth 1 obtain the initial states of the given automaton. For each of these nodes the flowpipes are computed and the reachable sets resulting from taking transitions whose guards are satisfied are passed to the next level in the corresponding location where the flowpipe computation iteratively restarts. In the picture, t_{l_i,l_j} denotes the discrete transition from location l_i to location l_j.

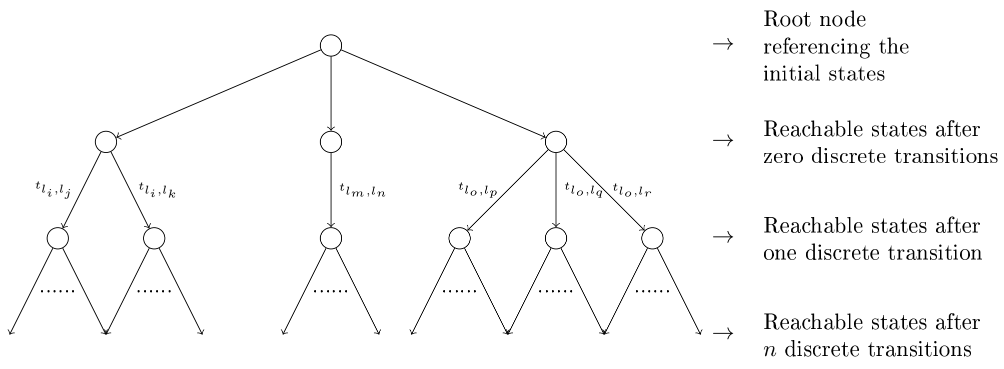

t_{l_i,l_j}
t_{l_i,l_k}
t_{l_m,l_n}
t_{l_o,l_p}
t_{l_o,l_q}
t_{l_o,l_r}
→ Root node referencing the initial states
→ Reachable states after zero discrete transitions
→ Reachable states after one discrete transition
→ Reachable states after n discrete transitions

Figure 3.1: A depiction of the tree-based reachability notion.

In order to keep the size of the tree compact, we do not store the computed flowpipes explicitly. Instead, we store the first segment which can be used to reconstruct the whole flowpipe using the context information of the node. The following section elaborates on this information. As a matter of fact, the current location, the current time step and representation are included. Hence, by the location we obtain its continuous dynamics that we can use to compute the flowpipe originating from the first segment using the granularity given by the time step and the representation according to the corresponding representation parameter.

3.2 Dynamic Backtracking

Using the scheme that was presented in the last section, we are able to employ backtracking in case the flowpipe of a node has a non-empty intersection with a given set of bad states. Figure 3.2 shows the abstract proceeding to achieve that. If during the flowpipe computation in a node, say n, such a non-empty intersection is detected, we backtrack to the node n_{Init} with depth 1 that n is derived from i.e. there exists a directed path from n_{Init} to n. In fact, we will even be able to backtrack only so far until we reach a node that has once been backtracked and use the knowledge of this previous backtracking. Once we have reached such a node, we refine the path from this node up to n. As we will see, the nodes also store context information of the reachability analysis that we can benefit from while we employ backtracking.

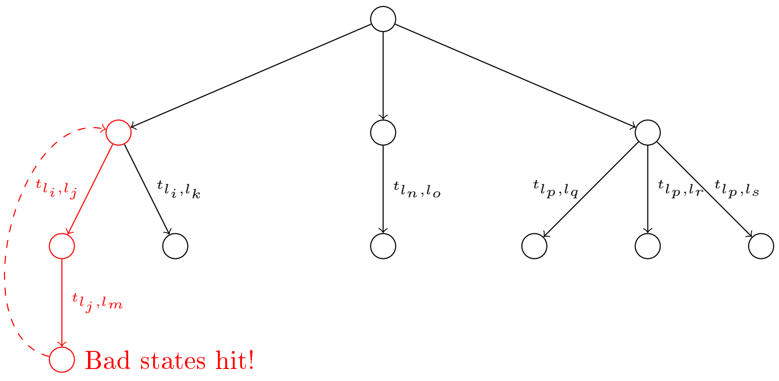


Figure 3.2: A depiction of the backtracking mechanism.

3.2.1 Dynamic Strategies

The aim of backtracking is to refine a path that has lead to intersecting one of the bad states. Therefore, we vary the utilized time step and representation. In order to specify which time step and representation are to be chosen when backtracking, we define a strategy $S = ((t_1,r_1),...,(t_n,r_n))$ with $t_i \in \mathbb{Q}$ and $r_i \in \{\text{Box}, \text{Zonotope}, \text{Support Function}, \text{Polytope}\}$ for $i \in \{1,...,n\}$. The initial parameter setup is (t_1,r_1). When the backtracking run with the parameter setting (t_i,r_i) failed, the next backtracking run is employed with the parameter setting (t_{i+1},r_{i+1}) for $i+1 \leq n$. If $i+1 > n$, the strategy was not able to resolve the bad states being intersected.

3.3 Context Information of Single Nodes

After having introduced the idea of the tree-based reachability and the backtracking mechanism that we employ, we will have a look at the information that each node comprises in order to see how we achieve these goals on a more technical level.

- ID: Each node has a unique ID. The ID id_n of a node n with depth i has length $i+1$ and is of the form $id_n = (0,a_1,...,a_i) \in \mathbb{N}^{i+1}$ specifying the path in the tree one has to take to reach n. Hence, to get from the root node to the next node, take the a_1-th child and take the a_2-th child of the obtained node to get to the next node and so on. This kind of ID is not only unique but also keeps track of the nodes' positions.

- References to the parent node and the childrens' nodes: For the backtracking mechanism that we employ, we need to be able to navigate through the tree. Therefore, each node holds references to its parent and its children.

- Location: The current location that the node computes the flowpipe for.

- Time step: As we aim to specify the reachability settings locally and not globally as in the traditional flowpipe-based approach, we store the time step that is used for the flowpipe computation in each node.

- Representation: The representation that is used for the flowpipe computation in a node.
- Initial set: The set initially obtained by taking a transition on the level before. As this set is modified during the computation of the flowpipe, we store it separately in order to later be able to recompute the flowpipe with new reachability settings.
- First segment: The first segment of the flowpipe in a node. As we do not store the flowpipe segments explicitly, we store the first segment in order to be able to reconstruct the flowpipe. Such a reconstruction might be needed in case we want to employ fixed-point recognition.
- Last segment: We also store the last determined segment of the flowpipe in a node. This provides us the possibility to stop the flowpipe computation and to resume it later.
- Guard satisfying intervals: When computing the flowpipe, we store for each pair of representation and transition the intervals in which the guards of the transitions are satisfied. In case, we recompute the flowpipe with the same representation and a finer time step, we can exploit this knowledge.
- Backtracking information: In case a node is refining a path i.e. it is a backtracking node, the backtracking information includes the path that is recomputed and the index $i \in \mathbb{N}$ keeping track of the position of the backtracking path such that we really only recompute the backtracking path. If during backtracking the representation stays equal and the time step gets finer, the backtracking information additionally includes the guard satisfying intervals of the backtracking path. Assume transition t in location l is enabled in the interval $[l_t,u_t]$ using boxes and a time step of 0.2. We can deduce that, using boxes with a time step of 0.1, being in l, the guard of t is satisfied the earliest at l_t and at most until u_t. Hence, we can omit intersecting the flowpipe segments for the time steps before l_t and after u_t.
- Strategy index: The index of the parameter setting in the strategy that was recently employed in this node. Initially it is set to zero. It is needed in order to correctly switch from one parameter setting to the next one regarding the chosen strategy and to recognize when all parameter settings have been applied.
- Only aggregation: For the case that we have non-aggregating transition, this flag stores for each node whether on the directed path from the root to it such transitions occurred. If so, it is $false$. Otherwise, it is $true$. We need this for the backtracking mechanism for non-aggregating transitions that is explained in detail in Section 3.6.

3.4 Implementation

In order to sum up and further formalize what was presented in this chapter up to now, in the following we give pseudocode versions of the backtracking mechanism and a refined version of the traditional flowpipe-based algorithm.

Algorithm 3 illustrates the backtracking mechanism. Before the proceeding is elaborated, we clarify the used data structures. Firstly, the backtracker exhibits a strategy. The strategy is a list of pairs of a time step and a representation as introduced before. Secondly, the backtracker keeps track of all backtracking runs that have already been executed. This is technically done by a map from location lists to an unsigned integer representing the index of the strategy element that was used in the last backtracking run.

The algorithm receives a node whose flowpipe intersected with the bad states. If this was the case with the final parameter setting of the strategy, see Line 1, we terminate as our strategy does not allow for further refinement. Otherwise, from Line 7 to 13 we backtrack until we have reached the root node and store the path that lead to intersecting the bad states just as we store the corresponding guard maps in case that only the time step changes. Afterwards, we check whether the backtracking path is a prefix of one of the previous backtracking runs. If not, in case the representation changes, we convert the initial set of the backtracking node b to the corresponding representation in Line 16. Then, we increase the strategy index, assign b with the new time step and representation and insert the new backtracking path with its strategy index in the backtracking history. Otherwise, we determine the longest such prefix and move to the node from which on we can exploit an old backtracking run in Line 25. Before returning the backtracking node, we enrich it by the backtracking information in Line 27. This includes the path that shall be recomputed, the guard maps and the index stating at which position of the backtracking path the node is which is initially 0.

Algorithm 3 Backtracking mechanism

```
Input: A node $n$ to be refined
Output: A node $b$ for the backtracking run
 1: if $n.stratIndex + 1 == strategy.size()$ then
 2:     $terminate()$
 3: end if
 4: $b := n$
 5: $onlyTimestepChanges =$
    $( n.representation == strategy.at( n.stratIndex + 1 ).representation ) \wedge$
    $( n.timeStep > strategy.at( n.stratIndex + 1 ).timeStep )$
 6: $path, guard\_maps := ()$
 7: while $b.depth \neq 0$ do
 8:     $path.pushFront( b.location )$
 9:     if $onlyTimestepChanges$ then
10:         $guard\_maps.pushFront( b.guard\_map )$
11:     end if
12:     $b := b.parent$
13: end while
14: if $\neg\exists((l_1, ..., l_i, ..., l_n), n) \in history. \forall j \leq i. l_j = path[ j ] \wedge n > b.stratIndex$
    then
15:     if $\neg onlyTimestepChanges$ then
16:         $b.initSet = convert( b.initSet, strategy )$
17:     end if
18:     $b.stratIndex{+}{+}$
19:     $b.timeStep = strategy.at( b.stratIndex ).timeStep$
20:     $b.representation = strategy.at( b.stratIndex ).representation$
21:     $history[ path ] = b.stratIndex$
22: else
23:     $prefix := longestPrefix( path )$
24:     $history[ path ] = history[ prefix ]$
25:     $b := moveTo( prefix, path, guard\_maps )$
26: end if
27: $b.backtrackingInfo = ( path, guard\_maps, 0 )$
28: $return\ b$
```

Algorithm 4 shows the pseudocode of the dynamic reachability analysis incorporating the previously presented backtracking mechanism. It exhibits a priority queue enabling us to prioritize nodes with a higher priority over those with a lower one instead of non-deterministically choosing them. Nodes that are not backtracking have a priority of 0. The priority of backtracking nodes corresponds to the length of their backtracking path. The intuition behind that is that nodes with long backtracking paths are preferred over those with shorter paths because they offer more possibilities for later backtracking runs to exploit them.

At first in Line 1, the priority queue is initialized with the passed initial nodes. Note that $Init_H$ contains more than just the initial sets as in the traditional approach i.e. it is already enriched by the node parameters that can be set initially e.g. the location, reference to the parent, the initial time step and so forth. While the queue is not empty, we process the nodes that are still left in it i.e. we take the node n

with the highest priority and call computeFlowpipe_dyn(n) in Line 4. This method works quite similar like the corresponding method of the traditional reachability analysis with the additional abilities needed for the dynamic analysis. That is, dealing with backtracking nodes and assigning nodes with the context information that they incorporate. If it processes a backtracking node, it only recomputes those paths that are on the critical path. In addition to that, if only the time step changed and became finer compared to the last parameter setting, computeFlowpipe_dyn(n) only recomputes segments for those pairs of timeStep and transition t for which also the more rough segmentation satisfied the guard of t. When a segment has a non-empty intersection with the bad states, the method returns true. Otherwise, it returns false as the flowpipe computation was completed and assigns the so-far empty parameters of the node like references to the children obtained by the taken transitions.

Algorithm 4 Dynamic Flowpipe-based Reachability Analysis

```
Input: A linear hybrid automaton $H$ with initial nodes $Init_H$ and a set of bad states
    $badStates$
1: $Q.enqueue( Init_H )$
2: while $\neg Q.empty()$ do
3:     $n := Q.dequeue()$
4:     $badStatesHit$ = computeFlowpipe_dyn($n$)
5:     if $badStatesHit$ then
6:         $backtrack(n)$
7:     end if
8:     computeJumpSuccessors($n$)
9: end while
```

3.5 Example

In this section, we will have a closer look at an example run using the previously introduced dynamic approach for reachability analysis. Therefore, the hybrid automaton depicted in Figure 3.3 will be taken as an input automaton. It features 6 locations l_0,...,l_5 with l_0 being the initial location and $x_0 \in$ [0,1] and $x_1 \in$ [0,10] being the initial valuation of the continuous variables x_0 and x_1. The chosen automaton shall illustrate the main features of the dynamic reachability analysis. After moving to l_1 from the initial location l_0, we expect intersecting the bad states, two times in l_2 and one time in l_3. The backtracking mechanism requires to change the representation and it is shown how a previous backtracking run can be reused. Furthermore, we have a part of the emerging reachability tree that does not require to employ backtracking, namely the one that emerges when taking the discrete transition from l_0 to l_5. Therefore, it shows the benefits of employing backtracking only for the part of the reachability tree that is part of a critical path. We compute an over-approximation of the reachable set up to a maximum of 4 discrete jumps and consider a time horizon of $T = 20$. The bad states are defined as $\{(l_2, x_1 \geq 21), (l_3, x_1 \geq 21)\}$. The employed strategy is ((0.5, Box), (0.1, Box), (0.1, Polytope)). All plots in this chapter and the whole thesis were created using *gnuplot* [WK13].

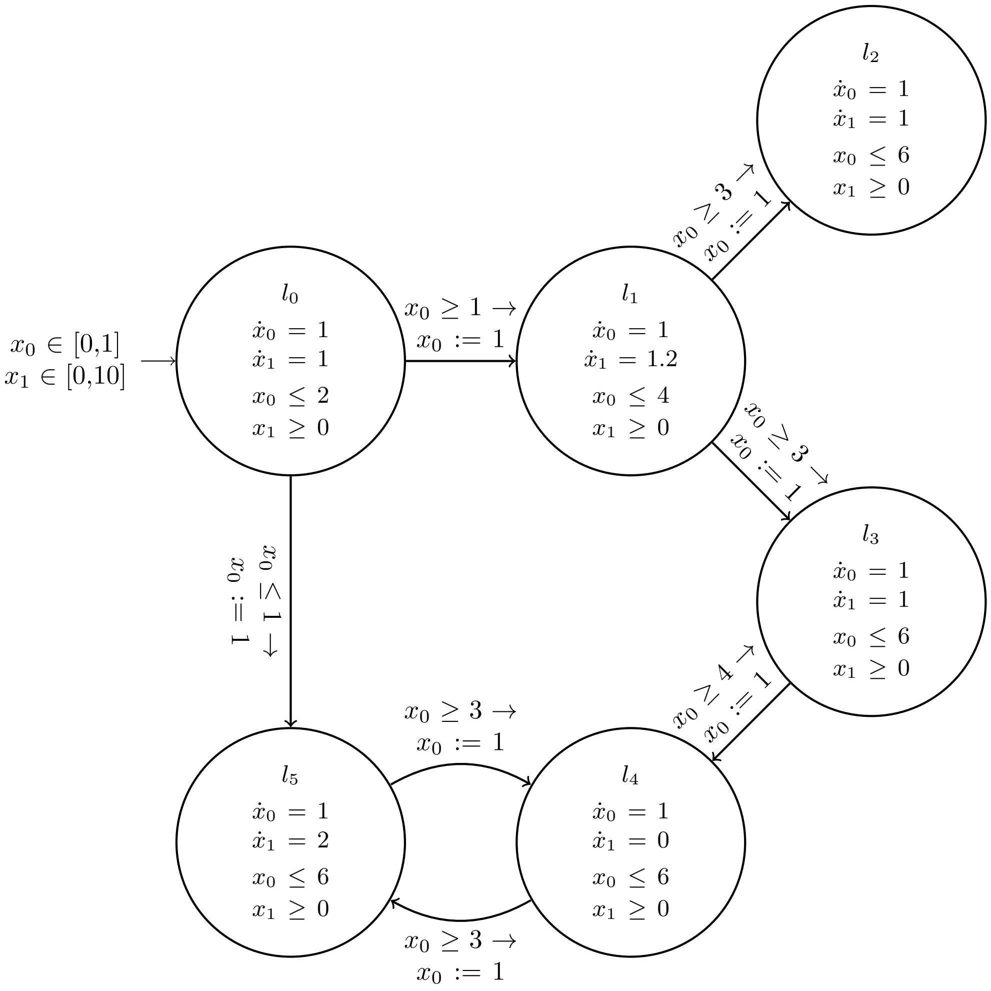


Figure 3.3: Graphical representation of the exemplary hybrid automaton.

At first, the initial valuations of x_0 and x_1 continuously develop according to the dynamics in l_0. For $x_0 \geq 1$ the outgoing discrete transition to l_1 can be taken and gets urgent when $x_0 = 2$ due to the invariant of l_0. As we perform reachability analysis, we compute the reachable set for the case that we stay in l_0 as long as the invariant is satisfied but also for the case that the transition is taken whenever the guard is satisfied. Thereby, we capture the whole reachable set of the given automaton. This procedure is continued analogously in the locations reachable from l_0. Figure 3.4 shows the dynamic of x_0, on the x-axis, and x_1, on the y-axis for the path $l_0 \rightarrow l_1 \rightarrow l_2$. We can observe that 3 flowpipes have been computed. The first one, in location l_0, ranges from $x_0 = 0$ to $x_0 = 2$. As the guard of the discrete transition from l_0 to l_1 is enabled for $x_0 \geq 1$ and sets x_0 to 1, the second flowpipe, in location l_1, begins at $x_0 = 1$ and is extended up to $x_0 = 4$. The third flowpipe, in location l_2, ranges from $x_0 = 1$ to $x_0 = 6$. During the analysis x_1 exceeds 21 in l_2 and therefore the bad states are intersected triggering the backtracking mechanism. A backtracking node for this path backtracks to the location l_0 and recomputes the flowpipes on the critical path with the second parameter setting. Due to the fact that the backtracking node refining the critical path $l_0 \to l_1 \to l_2$ has a higher priority, precisely 3 because of the length of the critical path, than the other nodes in the priority queue which have a priority of 0 because they are not backtracking nodes, the backtracking node is at the top of the priority queue and therefore preferred over the others. Because we only have a finer time step and the representation remains unchanged, we can exploit the guard maps of the nodes on this path that were stored in the previous run. E.g. for the discrete transition from l_1 to l_2 we can ignore all the time steps where $x_0 < 3$ and can omit the intersection computation of the corresponding flowpipe segments with the guard. Technically, two different time steps are not necessarily divisible by each other for which reason we also have do the intersection computation for the first time step before and the first time step after a transition guard is satisfied.

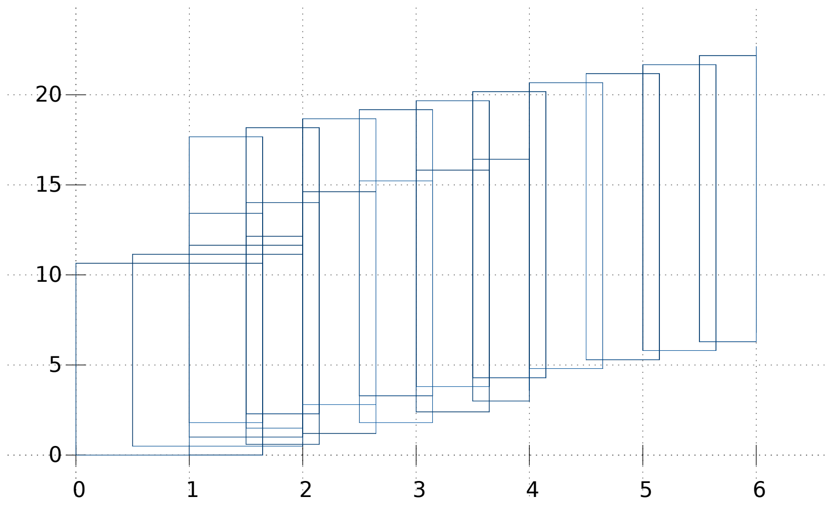


Figure 3.4: Continuous behavior for the initial reachability computation on the path $l_0 \to l_1 \to l_2$.

Figure 3.5 depicts the reachability tree at the point in time when the bad states are intersected in l_2. We can observe that the two discrete transitions i.e. from the initial location l_0 to l_1 respectively l_5 were taken just as on the next level the discrete transitions originating from l_1 respectively l_5. When we compute the flowpipe for the node in l_2, the bad states are intersected triggering the backtracking mechanism. As we can not reuse a previous backtracking run, we backtrack to the single initial node and refine the critical path $l_0 \to l_1 \to l_2$. Since the corresponding backtracking node, q_1 in Figure 3.5, has a higher priority than all other nodes in the queue i.e. q_2 and q_3 at this point in time, it is preferred over these nodes.

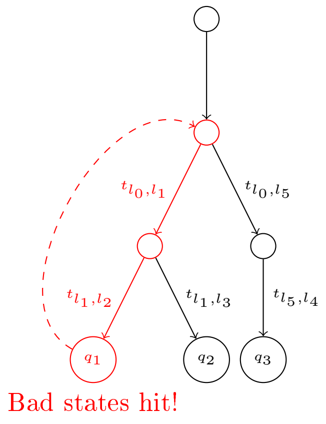


Figure 3.5: Reachability tree when the bad states are intersected for the first time in location l_2.

Figure 3.6 shows the variable valuations for the critical path with the new parameter setting next to the variable valuations of the previous run. Again, backtracking is employed, using polytopes and a time step of 0.1, as x_1 still exceeds 21. After this backtracking run the conflict is resolved due to the new, more precise, representation as we can see in Figure 3.7.

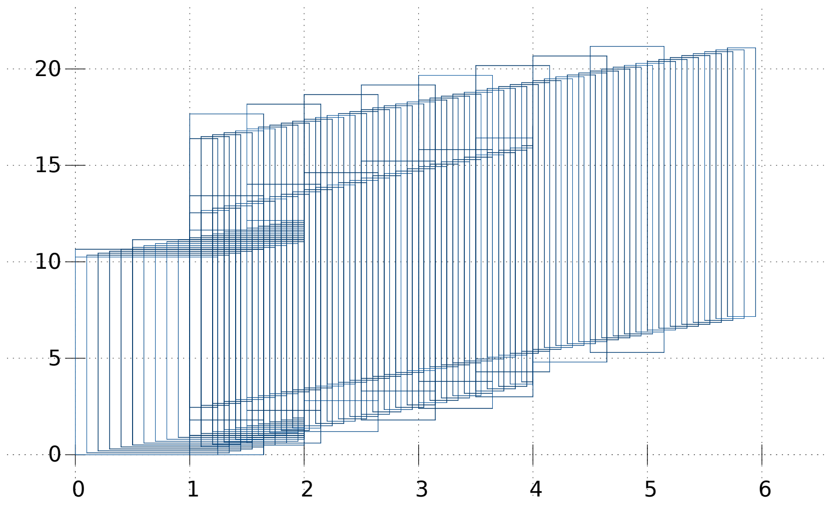


Figure 3.6: Continuous behavior for the first and for the second reachability computation on the path $l_0 \to l_1 \to l_2$.

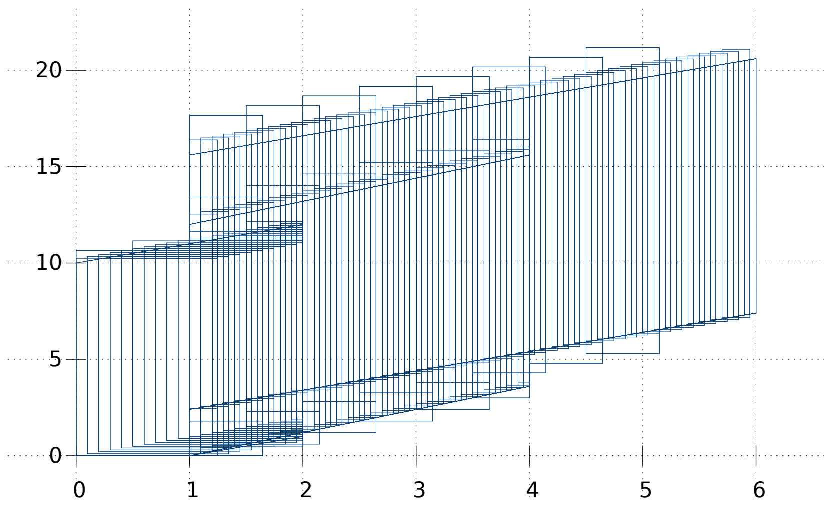


Figure 3.7: Continuous behavior for all 3 reachability computations on the path $l_0 \to l_1 \to l_2$.

Then, q_2 is next in the priority queue. Again, the bad states are intersected, this time in location l_3, because x_1 exceeds 21. Note that this node still operates with the initial strategy settings, using boxes and a time step of 0.5, as on its path the bad states were not hit so far. The reachability tree at the time when the bad states are intersected in location l_3, can be seen in Figure 3.8. As the previous conflict was resolved and there are no outgoing discrete transitions from l_2, the only remaining nodes in the priority queue are q_2 and q_3. Because the father node of q_2 has already been refined, we can reuse its initial set to refine the new critical path instead of backtracking to the initial node.

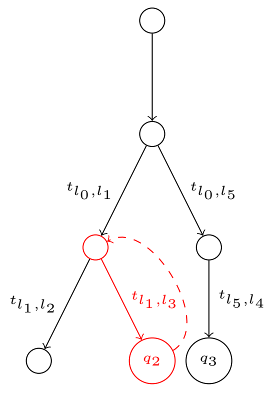


Figure 3.8: Reachability tree when the bad states are intersected in location l_3.

After this conflict was resolved, the reachability computation proceeds as usual.

3.6 Dynamic Approach with Non-Aggregating Transitions

While our approach primarily focuses on transitions that perform aggregation, see Subsection 2.5.2, we also offer a rough handling of non-aggregating transitions. In the following, we will elaborate on some of the differences that we have to deal with when transitions do not aggregate.

One difference is that we possibly have several backtracking nodes with the same location path and the same strategy index because the single segments in a single location that would be aggregated for aggregating transitions are not aggregated anymore. We handle this by backtracking for the first of these nodes and drop the following ones because the first one already computes the over-approximation that the following ones would also compute. This is technically done by the backtracking history that the backtracker administrates. It keeps track of all backtracking runs, including their paths and the corresponding strategy index.

Furthermore, the notion of reusing a former backtracking run is not that clear anymore when non-aggregating transitions occur as we can have several paths in our reachability tree with the same location sequence and strategy index. Therefore, we do not have a unique path that we can exploit for backtracking. Our current approach is that we use the *onlyAggregation* flag contained in each node stating whether from the initial node that n is derived by only aggregating transitions occur. If so, we can reuse a former backtracking path up to the last node, i.e. the one with the highest depth, where this flag is *true*.

Chapter 4

Experimental Results

After having introduced the idea and concepts of the dynamic reachability analysis, we will evaluate it by comparing it with the classical static reachability analysis. All tests have been executed on a *Thinkpad Edge 545* exhibiting 8GB of RAM and a four core processor where each core has a frequency of 2.4GHz. The implementation is part of a project called *HyDRA* being in a prototypical state of development.
The benchmarks used for testing and evaluating are partly taken from [hyb16]. Finally, we will discuss advantages and disadvantages of our approach. In the following, we adhere to the convention that, given a dynamic strategy $s = ((r_1,t_1),...,(r_n,t_n))$, s^{static} is given by $s^{static} = ((r_n,t_n))$. We use this in order to compare a dynamic strategy with the static approach. Assuming that the last parameter setting of a dynamic strategy is the most precise one, we need to employ this one in a static setting in order to ensure that we obtain the same result regarding whether the bad states were intersected.

4.1 Example Automaton

Firstly, we will evaluate the dynamic analysis on the example automaton of Section 3.5. For the sake of clearness, it is again depicted in Figure 4.1. As before, we choose a maximum of 4 discrete jumps, a time horizon of $T = 20$ and the bad states may be defined as $\{(l_2, x_1 \geq 21), (l_3, x_1 \geq 21)\}$. The employed strategy is $s = ((0.5, \text{Box}), (0.1, \text{Box}), (0.1, \text{Polytope}))$

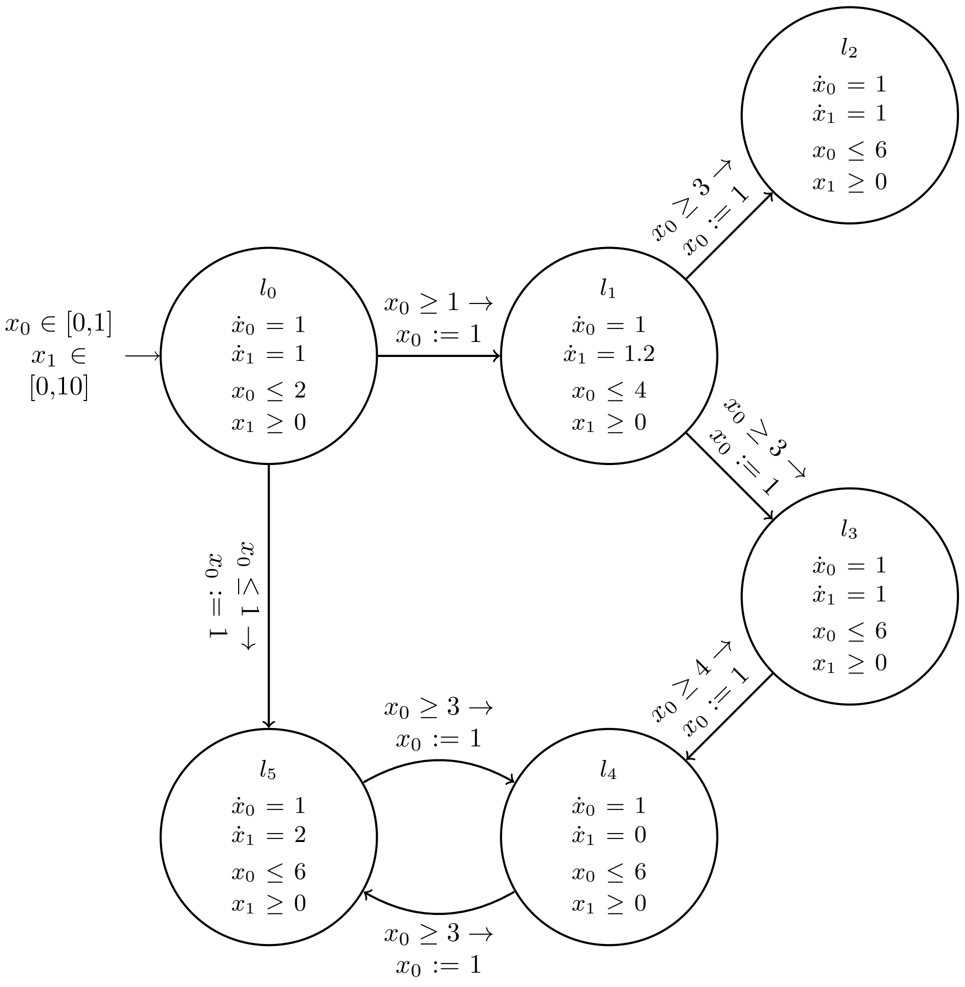


Figure 4.1: Graphical representation of the exemplary hybrid automaton.

The diagram below shows the mean value of 5 runs of each the dynamic analysis where s is the employed strategy, the static analysis s^{static} with the parameter setting where the time step is 0.1 and polytopes are used and the dynamic analysis with s where the discrete transitions from l_1 to l_2 and from l_5 to l_4 are non-aggregating.
We see that the dynamic analysis outperforms the static one. One of its advantages is that only those parts of the reachable set are precisely computed that tend to intersect the bad states. We can especially benefit from that on hybrid automata with different branches as the one we evaluate here. While on the paths starting with $l_0 \rightarrow l_1 \rightarrow l_2$

and $l_0 \to l_1 \to l_3$ polytopes with a time step of 0.1 are used due to backtracking, the paths starting with $l_0 \to l_5$ can be handled rather rough with boxes and a time step of 0.5.
For the sake of completeness, the diagram also shows the mean value of 5 runs of the dynamic analysis where the discrete transitions from l_1 to l_2 and from l_5 to l_4 are non-aggregating. We already addressed the trade-off between precision and complexity when using non-aggregating transitions. Clearly, this is only a single example but it indicates the trend also occurring on other benchmarks namely that the benefit of a higher precision rarely outweighs the disadvantage of increased complexity.

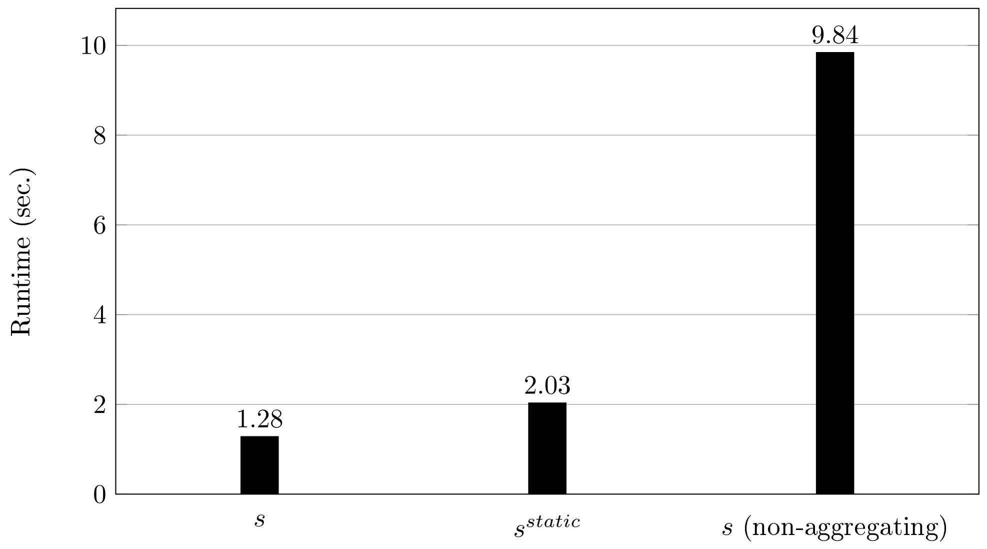
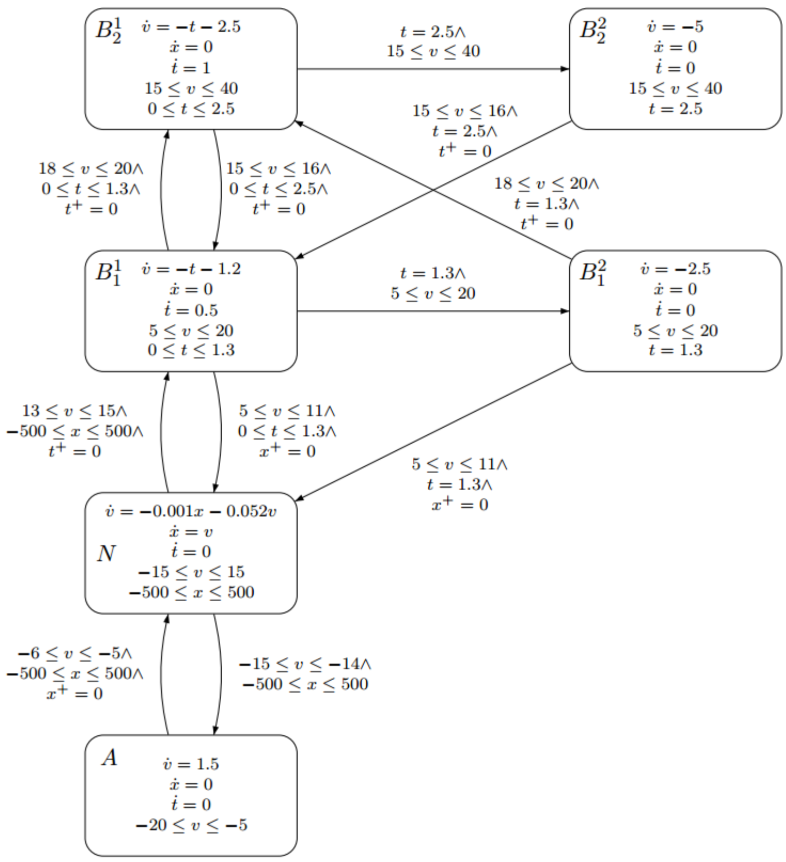


Figure 4.2: Runtime analysis using the example automaton with $s = ((0.5, \text{Box}), (0.1, \text{Box}), (0.1, \text{Polytope}))$, $s^{static} = ((0.1, \text{Polytope}))$ and s where the discrete transitions from l_1 to l_2 and from l_5 to l_4 are non-aggregating.

4.2 Cruise Control Model

In this section we use a cruise control model as a benchmark. It is used to model emergency brakes where v is the difference between the actual and the desired velocity, t the time and x an auxiliary variable. Precise information about this model can be found in [Oeh11]. The initial setting that we employed is in location B_2^1 with $x = 0$, $v \in [15{,}40]$ and $t \in [0{,}2.5]$. We choose a maximum of 4 discrete jumps, a time horizon of $T = 20$ and the bad states may be defined as $\{(l,\, x \geq 135) \mid l \in \{B_1^1, B_1^2, B_2^1, B_2^2, N, A\}\}$. Figure 4.3 gives a graphical depiction of the cruise control model that we use. In contrast to the example model from the last section, it includes variable's derivations that are non-constant making the continuous behavior of it more interesting. The values of v, on the x-axis, and x, on the y-axis, are visualized in Figure 4.4. The plot bases on a run where boxes with a time step of 0.1 were used.



B_2^1 $\dot{v} = -t - 2.5$ $\dot{x} = 0$ $\dot{t} = 1$ $15 \leq v \leq 40$ $0 \leq t \leq 2.5$
$t = 2.5\wedge$ $15 \leq v \leq 40$
B_2^2 $\dot{v} = -5$ $\dot{x} = 0$ $\dot{t} = 0$ $15 \leq v \leq 40$ $t = 2.5$
$15 \leq v \leq 16\wedge$ $t = 2.5\wedge$ $t^+ = 0$
$18 \leq v \leq 20\wedge$ $0 \leq t \leq 1.3\wedge$ $t^+ = 0$
$15 \leq v \leq 16\wedge$ $0 \leq t \leq 2.5\wedge$ $t^+ = 0$
$18 \leq v \leq 20\wedge$ $t = 1.3\wedge$ $t^+ = 0$
B_1^1 $\dot{v} = -t - 1.2$ $\dot{x} = 0$ $\dot{t} = 0.5$ $5 \leq v \leq 20$ $0 \leq t \leq 1.3$
$t = 1.3\wedge$ $5 \leq v \leq 20$
B_1^2 $\dot{v} = -2.5$ $\dot{x} = 0$ $\dot{t} = 0$ $5 \leq v \leq 20$ $t = 1.3$
$13 \leq v \leq 15\wedge$ $-500 \leq x \leq 500\wedge$ $t^+ = 0$
$5 \leq v \leq 11\wedge$ $0 \leq t \leq 1.3\wedge$ $x^+ = 0$
$5 \leq v \leq 11\wedge$ $t = 1.3\wedge$ $x^+ = 0$
N $\dot{v} = -0.001x - 0.052v$ $\dot{x} = v$ $\dot{t} = 0$ $-15 \leq v \leq 15$ $-500 \leq x \leq 500$
$-6 \leq v \leq -5\wedge$ $-500 \leq x \leq 500\wedge$ $x^+ = 0$
$-15 \leq v \leq -14\wedge$ $-500 \leq x \leq 500$
A $\dot{v} = 1.5$ $\dot{x} = 0$ $\dot{t} = 0$ $-20 \leq v \leq -5$

Figure 4.3: Graphical representation of the cruise control model [Oeh11].

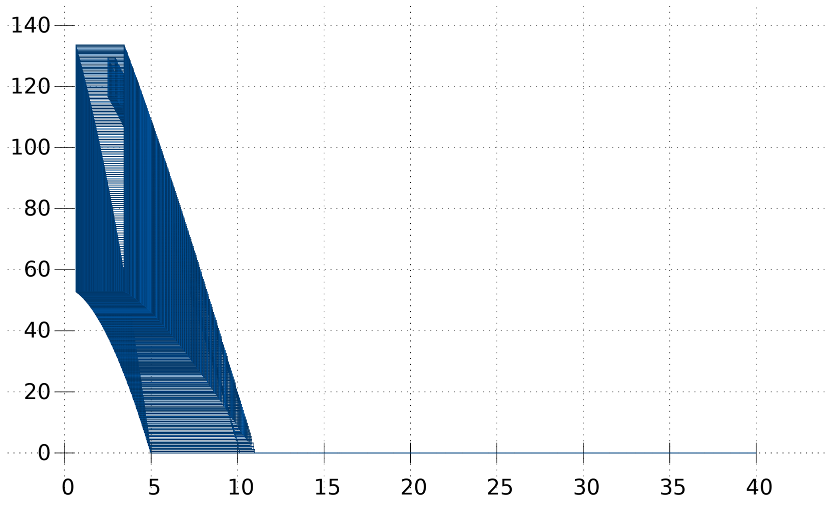


Figure 4.4: Continuous behavior of the cruise control model using boxes with a time step of 0.1. The x-axis depicts the values of v whereas the y-axis depicts the values of the variable x.

The diagram below shows the mean of 5 runs executed with each s_1 = ((0.4, Box), (0.1, Box)), s_2 = ((0.4, Box),(0.1, Support Function)) and s_3 = ((0.4, Box),(0.1, Polytope)) manifesting several characteristics of the dynamic analysis. Firstly, employing s_1, requiring 2 backtracking runs, is already faster than to use the corresponding static variant. In this case, the effort of using the dynamic approach pays off which is due to the fact that not all paths have to be considered with the finer time step and due to the fact that the guard satisfying intervals can be exploited during the backtracking runs. Although, applying s_1^{static} is quite fast here, the advantages of applying s_1 outweigh the extra effort going along with the overhead of performing the backtracking, updating the reachability tree and refining the old critical path. Applying s_2, which also needs 2 backtracking runs, confirms this trend. As support functions provide a more accurate over-approximation of the reachable set, the required effort of applying them globally becomes noticeable. Hence, the run where s_2 is applied, is faster than the corresponding static variant where for all paths of the cruise control model the more expensive representation with a quite small time step of 0.1 is applied. The comparison of s_3 and s_3^{static} underlines the observations made so far.

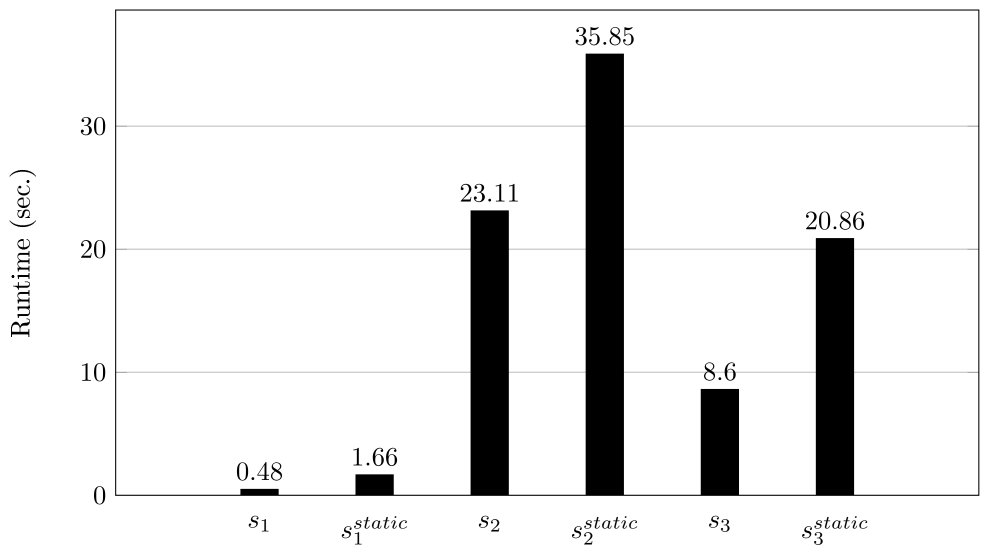
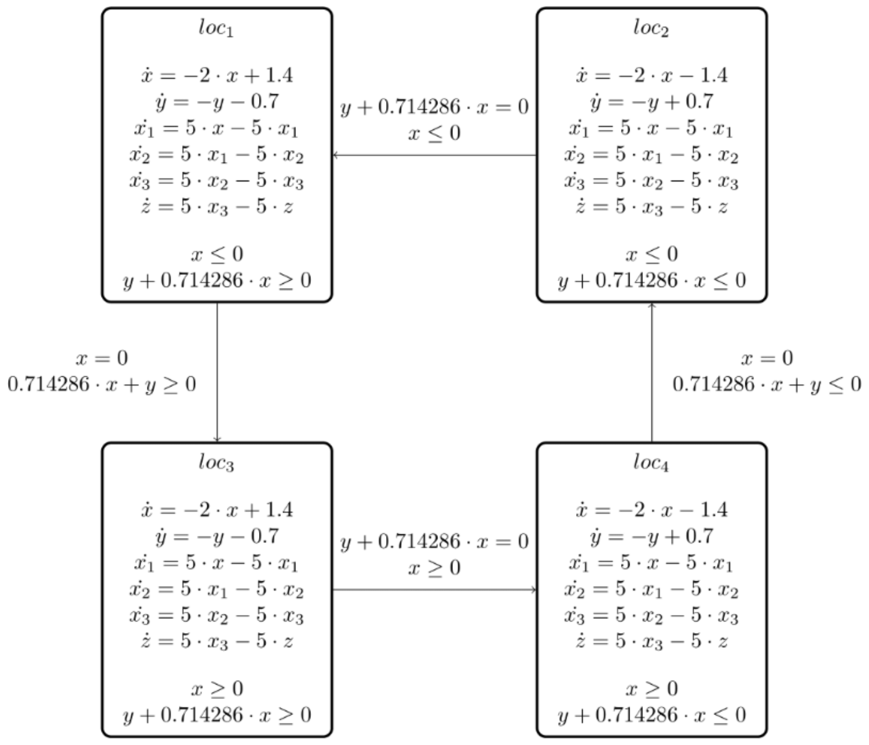


Figure 4.5: Runtime analysis using the cruise control model with s_1 = ((0.4, Box), (0.1, Box)), s_2 = ((0.4, Box), (0.1, Support Function)), s_3 = ((0.4, Box), (0.1, Polytope)) and their corresponding static versions s_1^{static} = ((0.1, Box)), s_2^{static} = ((0.1, Support Function)) and s_3^{static} = ((0.1, Polytope)).

4.3 Filtered Oscillator Model

In this section we use a filtered oscillator model as a benchmark. It is used to model a switched oscillator for the variables x and y. The variables x_1,x_2,x_3 and z filter the signal x and z is the corresponding output. Precise information about this model can be found in [hyb16]. The initial setting that we employed is in location loc_3 with $x \in [0.2,0.3]$, $y \in [-0.1,0.1]$ and $x_1,x_2,x_3,z = 0$. We choose a maximum of 5 discrete jumps, a time horizon of $T = 4$ and the bad states may be defined as $\{(l,\ y \geq 0.5) \mid l \in \{loc_1,loc_2,loc_3,loc_4\}\}$. Figure 4.6 gives a graphical depiction of the filtered oscillator model that we use. The values of x, on the x-axis, and y, on the y-axis, are visualized in Figure 4.7. The plot bases on a run where boxes with a time step of 0.01 were used.


loc_1
$\dot{x} = -2 \cdot x + 1.4$
$\dot{y} = -y - 0.7$
$\dot{x_1} = 5 \cdot x - 5 \cdot x_1$
$\dot{x_2} = 5 \cdot x_1 - 5 \cdot x_2$
$\dot{x_3} = 5 \cdot x_2 - 5 \cdot x_3$
$\dot{z} = 5 \cdot x_3 - 5 \cdot z$
$x \leq 0$
$y + 0.714286 \cdot x \geq 0$
$y + 0.714286 \cdot x = 0$
$x \leq 0$
loc_2
$\dot{x} = -2 \cdot x - 1.4$
$\dot{y} = -y + 0.7$
$\dot{x_1} = 5 \cdot x - 5 \cdot x_1$
$\dot{x_2} = 5 \cdot x_1 - 5 \cdot x_2$
$\dot{x_3} = 5 \cdot x_2 - 5 \cdot x_3$
$\dot{z} = 5 \cdot x_3 - 5 \cdot z$
$x \leq 0$
$y + 0.714286 \cdot x \leq 0$
$x = 0$
$0.714286 \cdot x + y \geq 0$
$x = 0$
$0.714286 \cdot x + y \leq 0$
loc_3
$\dot{x} = -2 \cdot x + 1.4$
$\dot{y} = -y - 0.7$
$\dot{x_1} = 5 \cdot x - 5 \cdot x_1$
$\dot{x_2} = 5 \cdot x_1 - 5 \cdot x_2$
$\dot{x_3} = 5 \cdot x_2 - 5 \cdot x_3$
$\dot{z} = 5 \cdot x_3 - 5 \cdot z$
$x \geq 0$
$y + 0.714286 \cdot x \geq 0$
$y + 0.714286 \cdot x = 0$
$x \geq 0$
loc_4
$\dot{x} = -2 \cdot x - 1.4$
$\dot{y} = -y + 0.7$
$\dot{x_1} = 5 \cdot x - 5 \cdot x_1$
$\dot{x_2} = 5 \cdot x_1 - 5 \cdot x_2$
$\dot{x_3} = 5 \cdot x_2 - 5 \cdot x_3$
$\dot{z} = 5 \cdot x_3 - 5 \cdot z$
$x \geq 0$
$y + 0.714286 \cdot x \leq 0$
Figure 4.6: Graphical representation of the filtered oscillator model [hyb16].

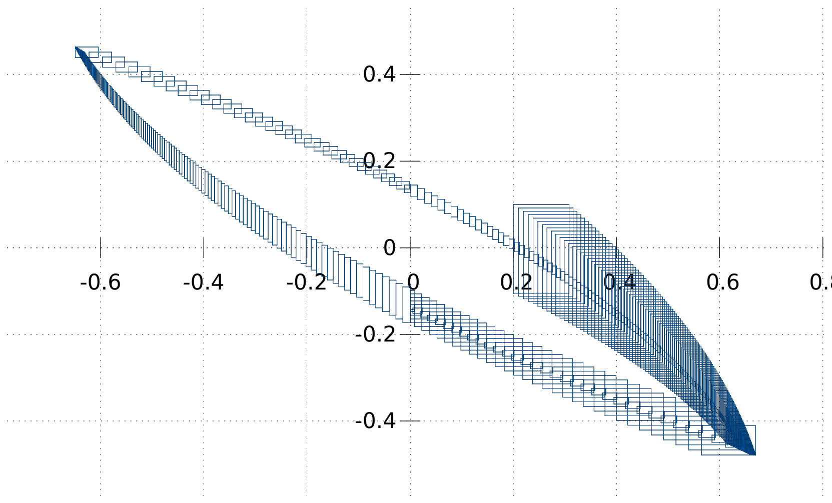


Figure 4.7: Continuous behavior of the filtered oscillator model using boxes with a time step of 0.01. The x-axis depicts the values of x whereas the y-axis depicts the values of the variable y.

The diagram below shows the mean of 5 runs executed with each $s_1 = ((0.5, \text{Box}), (0.1, \text{Box}), (0.05, \text{Box}))$, $s_2 = ((0.5, \text{Box}), (0.1, \text{Box}), (0.01, \text{Box}))$ and their static variants s_1^{static} and s_2^{static}. The employed input automaton is interesting in the sense that its discrete structure is linear i.e. for each pair of locations we have exactly one discrete path between them. Firstly, employing s_1, requiring 2 backtracking runs on the same path i.e. on this path we employ boxes with a time step of 0.05 in the second backtracking run, is more slowly than employing s_1^{static}. Therefore, this automaton serves as an example for the fact that the effort that we put into backtracking does not always pay off. Because of the linear discrete structure of the input automaton, we can not benefit of computing some branches rather roughly and other ones precisely. Still, for such an automaton we can exploit the guard satisfying intervals. Employing s_2 indicates that this can suffice to outperform s_2^{static} for a very fine time step of 0.01. Here, the fact that we can exploit the guard satisfying intervals during backtracking avoids that we have to intersect a huge number of flowpipe segments with the corresponding transition guards for which we can already deduce that the intersection is empty. Hence, in case we can omit a lot of these intersection computations, the dynamic analysis has an advantage over the static one.

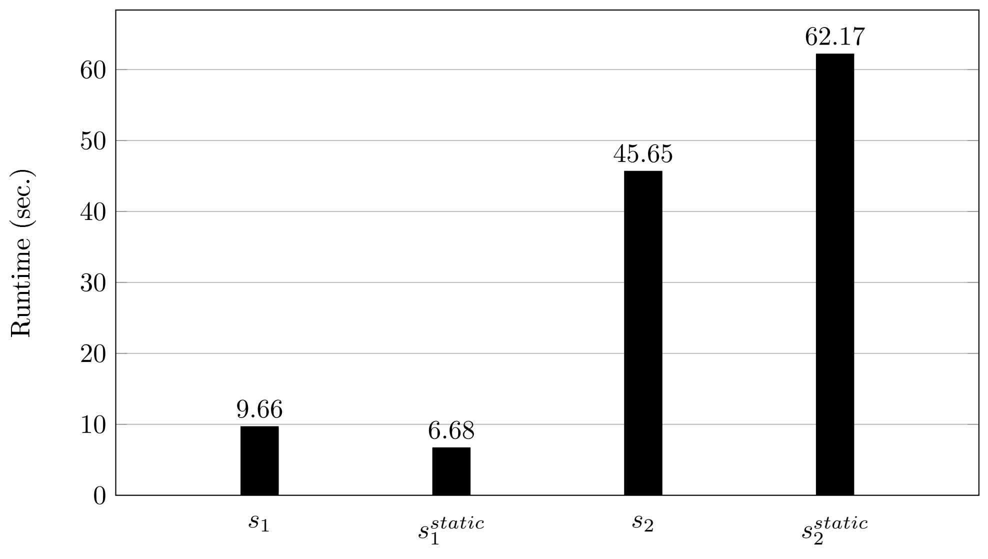


Figure 4.8: Runtime analysis using the filtered oscillator model with $s_1 = ((0.5, \text{Box}), (0.1, \text{Box}), (0.05, \text{Box}))$, $s_2 = ((0.5, \text{Box}), (0.1, \text{Box}), (0.01, \text{Box}))$ and their corresponding static versions $s_1^{static} = ((0.05, \text{Box}))$ and $s_2^{static} = ((0.01, \text{Box}))$.

4.4 Strategy Choice

Finally, we will conclude on the results of the last section and discuss how good strategies for a dynamic reachability analysis might be designed. Deducing from the benchmarks that were presented, boxes are quite fast in lower dimensionalities. For automata where the continuous behavior is not in such a way that we make huge approximation errors when using boxes, the tests suggest that it might be a good idea to start with boxes and a time step of about 0.2 to about 0.4. Choosing a time step that is too high initially bears the danger that guards of transitions are satisfied that would not be satisfied when using a finer time step. Therefore, we would might follow paths that we could actually drop. For the case that this setting fails, one could proceed with boxes and a relatively fine time step of 0.01 to 0.1 which is still relatively fast compared to applying e.g. polytopes with a fine time step. If this again fails, the continuous behavior of the linear hybrid automaton is possibly in such a manner that we need a more precise representation like support functions or polytopes using a fine time step which can be made even smaller once more.
As the number of benchmarks for linear hybrid systems is quite limited at the moment, it is hard to make general statements about globally good strategies but from the benchmarks that were used for testing and evaluating our approach the aforementioned suggestions turned out to be quite successful.

4.5 Summary

The benchmarks indicate that using our dynamic reachability approach can make reachability analysis more flexible and faster. Because it adds some overhead, the models to be considered need to have a certain level of complexity such that this overhead pays off. There are several factors influencing the success of our approach. Firstly, the way in which the bad states are defined. In case a rather rough analysis already suffices in order to determine whether they are intersected, the overhead of the dynamic analysis might not pay off. Another factor is the branching degree of the input automaton. Consider the example automaton from Section 3.5 featuring two discrete transitions from the initial location l_0 to l_1 respectively l_5 which therefore has a rather compositional structure. While the paths of the reachability computation with the infix $l_0 \rightarrow l_5$ can be computed rather rough as they do not intersect the bad states, the paths with the infix $l_0 \rightarrow l_1$ have to be computed rather fine-grained in order to not intersect the bad states. In order to avoid intersecting the bad states in a static setting, one would have to compute all paths of this automaton with a fine-grained setting. In contrast, the filtered oscillator model from Section 4.3 shows that automata with a low branching degree do not exhibit this benefit. On the other hand we can still reuse the results of previous computations for those automata. A further advantage of the dynamic approach is its flexibility. In case one wishes to compute a fine-grained over-approximation of the reachable set for all parts of a given model, one can simply define a strategy where already the initial parameter setting is chosen correspondingly.

Chapter 5

Conclusion

This thesis aimed at developing an approach for dynamic reachability analysis for linear hybrid automata that overcomes weaknesses of classic static approaches. Thereby, we reuse results of previous computations and are enabled to compute finer over-approximations only for those parts of the reachable set that tend to intersect a given set of bad states.

5.1 Summary

After having presented the theoretical background, we provided a basis for our dynamic reachability approach. Therefore, we introduced a new data structure, the reachability tree, including context information for each node that enabled us to achieve our goals. Firstly, it stores information that is needed for backtracking and thereby recomputing certain paths of the reachability tree with new parameter settings. Additionally, it keeps track of previous computation results and enables us to recompute flowpipes in case this is needed e.g. for fixed-point recognition. We then established the backtracking mechanism and gave a pseudocode implementation of it. After this, we completed the circle by showing how the common (static) reachability algorithm has to be adapted in order to incorporate the notions of our dynamic approach. An example run then illustrated the main ideas of the previously introduced concepts.
Finally, we evaluated our algorithm. The benchmarks suggest that it can make reachability analysis of linear hybrid automata more feasible. Definitely, employing the backtracking mechanism introduces a certain overhead. Hence, for quite small automata regarding their discrete structure this overhead might not payoff. But once the considered automata become more complex in terms of their discrete structure, the effort that we put into establishing a dynamic analysis tends to outweigh the additional overhead.

5.2 Future Work

In this section we will have a brief look at possible optimizations and extensions that the dynamic analysis could benefit of.

5.2.1 Fixed-Point Recognition

As pointed out in [SAC+15], fixed-point recognition is a challenge in the verification of hybrid systems. That is, we want to check whether newly obtained reachable sets are already contained in our global set of collected reachable sets. The reachability tree provides a solid basis for that. A possible approach to achieve this, would be to traverse the reachability tree. Thereby, one would reconstruct the flowpipes of those nodes having the same location as the reachable set for which the fixed-point check is executed for and check whether these flowpipes contain the set. As this bears the danger of getting quite expensive, one could introduce an additional parameter for each node storing an over-approximation of its flowpipe. Then, one would only have to reconstruct the whole flowpipe in case the reachable set to be tested is contained in this over-approximation.

5.2.2 Parallel Dynamic Approach

We might benefit from parallelizing our approach. When executing it sequentially, the time dependencies between different nodes in the working queue are met. In case we aim to execute it in parallel we would have to make sure that this still is the case. Consider e.g. two backtracking runs such that the second one would exploit the first one. In such a scenario, we would have to make sure that the first node has recomputed the critical path far enough before the second node moves on with its execution. This and other synchronization issues would have to be resolved in order to parallelize our method.

5.2.3 Extending the Strategy Settings

Current strategies enable us to vary the time step and the representation. As a matter of fact, one could extend this. Reconsider the example automaton from Section 3.5. The backtracking run on the critical path $l_0 \rightarrow l_1 \rightarrow l_3$ exploited the previous backtracking run on the path $l_0 \rightarrow l_1 \rightarrow l_2$ and therefore polytopes were used because for the latter path the second parameter setting, boxes with a time step of 0.1 did not suffice to resolve the conflict. However, for the locations reachable from l_3, the second parameter setting would suffice in terms of intersecting the bad states. Therefore, one could convert the initial sets that we obtain by taking the discrete transition from l_3 to l_4 from polytopes to boxes. Additionally, one could allow to stop backtracking before an initial node has been reached and recompute the path from there on. In case this does not resolve the conflict, we might have multiple re-computations of flowpipes when we subsequently backtrack further just as this limits the potential of reusing previous backtracking runs but there may be scenarios in which it could be beneficial.

5.3 Conclusion

Reachability analysis of (linear) hybrid automata is an emerging field of research. Our dynamic approach can make this analysis more flexible and feasible. Using it, we benefit from computing fine over-approximations only for those paths that tend to intersect the bad states just as we reuse previous computation results. The suggested extensions might further leverage it. As the bandwidth and depth of available verification techniques in this field will grow, it will be interesting to monitor how they compete.

Bibliography

[Á15] Erika Ábrahám. Modeling and analysis of hybrid systems, Summer term 2015.

[ACHH93] Rajeev Alur, Costas Courcoubetis, Thomas A Henzinger, and Pei-Hsin Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid systems*, pages 209–229. Springer, 1993.

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

[Dan00] Thi Xuan Thao Dang. *Vérification et synthese des systemes hybrides.* PhD thesis, 2000.

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

[hyb16] `https://ths.rwth-aachen.de/research/projects/hypro/benchmarks-of-continuous-and-hybrid-systems/`, September 2016.

[Leo96] IE Leonard. The matrix exponential. *SIAM review*, 38(3):507–512, 1996.

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

[Oeh11] Jens Oehlerking. *Decomposition of stability proofs for hybrid systems.* PhD thesis, Universität Oldenburg, 2011.

[SAC$^+$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 *Proceedings of the 5th Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*, volume 9361 of *Information Systems and Applications, incl. Internet/Web, and HCI*, pages 8–24. Springer, 2015.

[WK13] Thomas Williams and Colin Kelley. Gnuplot 4.6: an interactive plotting program. `http://gnuplot.sourceforge.net/`, April 2013.

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