Theory of Hybrid Systems Informatik 2
RWTH AACHEN UNIVERSITY

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

MASTER OF SCIENCE THESIS

INTEGRATION OF HYBRID SYSTEMS VERIFICATION METHODS IN MATLAB

Marta Grobelna

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

Additional Advisor:
Stefan Schupp, M.Sc.

Aachen, 16.09.2019

Abstract

Hybrid systems are systems that exhibit continuous and discrete behaviour. Cyber-physical systems for instance, in which digital controllers interact with a continuous environment are a common example of a hybrid system. Since such systems are often safety critical, a proper verification of their correctness is needed. Hybrid automata, as an abstraction of a hybrid system, are used as a model for formal verification via reachability analysis.

There are various tools that implement reachability analysis algorithms for hybrid automata. In industry tools such as MATLAB are used for prototyping and development. The tool CORA, which is implemented in MATLAB, has shown in various industrial applications that this is a sensible criterion for industrial developers. In the context of this thesis a MATLAB-wrapper of HYPRO was developed. Our evaluation shows that the increased cost of wrapping HYPRO can be mitigated by the increased speed of the underlying C++ implementation of HYPRO.

Acknowledgements

First of all I want to thank Prof. Dr. Erika Ábrahám for the opportunity to write this master thesis. I want to thank my supervisor Stefan Schupp for the continuous support and feedback. Many thanks to Aleksandra for the continuous mental support, to Isabella and Albret for always being there for me, and to my friends, Niklas and Nils, for their wise advice and the support especially during the last weeks. Finally, I want to express my special thanks to my parents who made all this possible and always believed in me.

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.

Marta Grobelna
Aachen, den 16. September 2019

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

Contents

Chapter 1

Introduction

Systems of sensors and controllers are called *embedded control systems (ECS)*. Their main area of application are so called *cyber-physical systems (CPSs)* where multiple embedded computers communicate via a network to control and monitor a physical environment. Examples of such systems are modern transportation systems, or medical and factory equipment. A set of sensors can monitor a number of different physical quantities, such as temperature, pressure, density, velocity, etc. This information can be used to recognize dangerous situations, such as overheating of a liquid, leak in a tank, or a collision. In order to counter react in case of such situation, actuators can be used to change the physical environment and so avoid an accident, or at leas minimize the damage. Therefore, CPSs can improve our safety, minimize the number of system failures, and decrease the repair costs if an accident cannot be avoided. Moreover, they can increase the efficiency, as the number of interrupts caused by failures can be decreased, and a number of tasks originally executed by humans can be automated and therefore often accelerated. This improvements, however, come with the price of high expectations on CPSs. It is expected that the systems work reliably in every situation even though it is expected that the tasks executed by them get more and more complex [CPPAV06, LS16].

In order to meet this expectations, the number of sensors and controllers in CPSs increases constantly. The effects of the increased intelligence of such systems can be observed on the example of vehicles. According to World Health Organization (WHO) 1.25 Million people die every year in a traffic accident and about 50 Million get injured. Although this numbers may appear huge, in Germany, as well as in other Western nations, the number of deaths has fallen by 66 percent since 1993, whereat the distance traveled has increased by 23 percent. The reason for this tendency are cars which are provided with intelligent driving assistance systems, which aim to minimize human failures [dAe15, Flo93].

Since sensors forward the information to a computer that can process the information and send further information to an actuator that can properly react to changes in an environment, the digital (discrete) world directly interacts with the physical (continuous) world. Such systems are called *hybrid systems*, as they exhibit continuous and discrete behaviour. The modelling and verification of hybrid systems is challenging, as both behaviours have to be considered jointly in order to capture all possible behaviours of the system. However, in context of CPS, discrete and continuous behaviours have different properties. On the one hand, discrete systems are usually

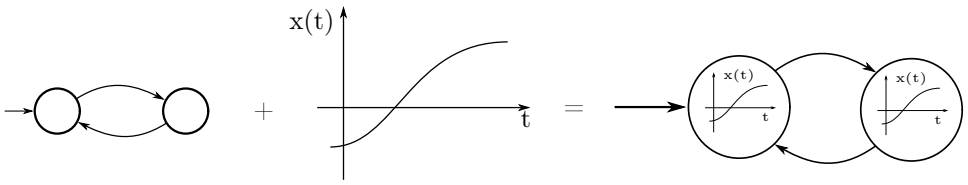


Figure 1.1: Hybrid systems combine discrete and continuous dynamics. This illustration is taken from [Ábr12].

predictable and deterministic as they proceed according to a sequence of clearly defined instructions. On the other hand, continuous behaviour is far less predictable as the physical environment is influenced by a number of external factors which can cause unpredictable behaviour of the system [dAe15].

Since CPSs contain an enormous number of details, a direct analysis of their behaviour is infeasible. Therefore, an abstract model of the system is needed that can disregard unnecessary details. Subsequently *formal methods* can be used to analyze the properties of interest of the model [Alu11, DLV11].

Another important point is the fact that the time a CPS needs to solve a certain task is not a question of performance but, in most cases, a question of correctness. Therefore, if a CPS needs too long time to react to a certain circumstance, this behaviour is rated as a system failure [dAe15].

Last important point about CPSs, is the fact that physical processes usually are a sum of multiple different sub-processes. Therefore, CPSs are usually compositional, i.e., they consist of a network of embedded computers where each is responsible for a different sub-process. This additionally increase the complexity as the information captured by one sub-process may also be important for another sub-process, thus, the components need to communicate in order to exchange the information. Moreover, the delivery times of the information may also be time-critical [DLV11].

All the challenges that CPSs have to overcome, induced the need of efficient and reliable verification methods in order to proof the correct functionality. One possible way to proof the correctness of such systems is via formal verification of the hybrid automaton that models the system. Mathematically, discrete systems can be modeled by *transition systems*, while continuous behavior can be described using *ordinary differential equations (ODE). Hybrid automata* combine the two modelling paradigms [Alu11]. A hybrid automaton is an extended finite state transition system where each location corresponds to a different discrete state of the system. Moreover, each location is augmented with dynamic behaviour for that state, given as an ODE. Figure 1.1 illustrates the idea. A formal definition of hybrid automata will be presented in the next chapter.

Usually, for each CPS there exists a set of specifications that must be fulfilled. These specifications define a set of *bad states*, i.e., states the hybrid automaton should not reach. Hence, executions of an automaton that models a correctly working system, never reaches any bad state [Alu11].

The idea of safety specifications and bad states can be illustrated on the example of an airbag system. A correctly working airbag system will always open the airbags on time, disregarding how fast the driver was accelerated against the steering wheel. An exemplary bad state is the state where the driver hits the steering wheel and the airbag

is closed. Another bad state is the situation when the airbag was opened although the car did not crash. In order to prove that these two situations never happen, one can prove that the hybrid automaton, modelling the airbags system, never reaches the two bad states. Verification of such specifications can be done via *reachability analysis* and will be explained detailed in the next chapter.

Chapter 2

Preliminaries

The aim of this chapter is the introduction of preliminary information needed for the rest of this thesis. The chapter consists of three sections. In the first part of this chapter, the syntax and semantics of hybrid automata will be introduced. The second section deals with the reachability analysis of hybrid systems. In the last part of this chapter the most prominent state set representations as well as the most important set operations will be introduced.

2.1 Modelling Hybrid Systems

Hybrid automata are used to model the behaviour of hybrid systems. They extend transition systems [BK08] by continuous transitions. The following definition specifies the syntax of hybrid automata

Definition 2.1.1 (**Syntax of Hybrid Automata** ([ACH+95]))**.** *A* hybrid automaton ($\mathcal{H}$) *is a tuple* $\mathcal{H} = (Loc, Var, Lab, Trans, Flow, Inv, Init)$ *consisting of seven components that are defined as follows.*

- Loc *is a finite set of* locations.
- Var *is a finite set of real-valued* variables. *A* valuation ν *is a function that assigns a real-value* $\nu(x) \in \mathbb{R}$ *to each variable* $x \in Var$*. The set of all valuations is denoted by* V*.*

  A state *is a pair* (l,ν) *consisting of a location* $l \in Loc$ *and a valuation* $\nu \in V$*. The set of all states is denoted by* Σ*.*
- Lab *is a finite set of* synchronization labels *that contains the* stutter *label* $\tau \in Lab$*.*
- Trans *is a finite set of edges called* transitions*. Each transition* $t = (l, \alpha, \mu, l')$ *consists of a* source *location* $l \in Loc$*, a* target *location* $l' \in Loc$*, a synchronization label* $\alpha \in Lab$*, and a power-set* $\mu \subseteq V^2$ *that defines resets and guards. A transition is* enabled *if for the current valuation of variables* ν *there exists a* $\mu \in V^2$ *such that* $\mu = (\nu, \nu')$*. The current valuation* ν *is then reset to* ν' *after taking the transition. Moreover, it is required that for each location* $l \in Loc$*, there is a* stutter transition *of the form* (l, τ, Id, l) *where* $Id = \{(\nu, \nu) \mid \nu \in V\}$*.*

- Flow *is a labeling function that assigns a set of* flows *to each location* $l \in Loc$. *Each flow is a function from the non-negative reals* $\mathbb{R}^{\geq 0}$ *to* V. *It is required that the flows of each location are* time-invariant, *i.e., for all location* $l \in Loc$, *flows* $f \in Flow(l)$, *and non-negative reals* $t \in \mathbb{R}^{\geq 0}$, *also* $(f+t) \in Flow(l)$, *where* $(f+t)(t') = f(t+t')$ *for all* $t' \in \mathbb{R}^{\geq 0}$.
- Inv *is a labeling function that assigns an invariant* $Inv(l) \subseteq V$ *to each location* $l \in Loc$.
- Init$\subseteq \Sigma$ *is the set of initial states.*

An execution step, denoted by $\rightarrow$, is either a discrete or a continuous step, since hybrid automata can exhibit discrete and continuous behaviour. The two types of steps are defined by the semantics of a hybrid automaton.

Definition 2.1.2 (**Semantics of Hybrid Automata** ([ACH+95]))**.** *The semantics of a hybrid automaton* $\mathcal{H} = (Loc, Var, Lab, Trans, Flow, Inv, Init)$ *is given by an operational semantic consisting of two rules, one for the discrete instantaneous steps and one for the continuous time steps.*

1. *Discrete step semantics*

$$\frac{(l,\alpha,\mu,l') \in Trans \;\; (\nu,\nu') \in \mu \;\; \nu \in Inv(l) \;\; \nu' \in Inv(l')}{(l,\nu) \xrightarrow{\alpha} (l',\nu')} Rule_{discrete} \tag{2.1}$$

2. *Time step semantics*

$$\frac{f \in Flow(l) \;\; f(0) = \nu \;\; f(t) = \nu' \;\; t \geq 0 \;\; f([0,t]) \subseteq Inv(l)}{(l,\nu) \xrightarrow{t} (l,\nu')} Rule_{time} \tag{2.2}$$

A *discrete step*, denoted by $\xrightarrow{\alpha}$, corresponds to taking a discrete transition $\epsilon = (l,\alpha,\mu,l') \in Trans$, so the control location and the variables valuation are updated as defined by μ. A *time step*, denoted by $\xrightarrow{\tau}$, changes the valuation of the variables according to the flow of the current location. A discrete step can only be taken if the current values of the variables can satisfy the guard of the desired transition. A time step can only be accomplished if the invariant of the current location is satisfied by the current variables valuation, the valuation after taking the time step and all the time between the two points in time [ACH+95].

A *run* π of a hybrid automaton $\mathcal{H}$ is a sequence of states $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \rightarrow \ldots$ where $\sigma_i = (l_i, \nu_i), l_i \in Loc, \nu_i \in V$ for $i \geq 1$ and $\sigma_0 = (l_0, \nu_0),\ l_0 \in Init,\ \nu_0 \in Inv(l_0)$. A state σ is *reachable* in $\mathcal{H}$ if and only if there exists a run of $\mathcal{H}$ starting in an initial state of $\mathcal{H}$ and leading to σ [ACH+95].

A run π is *time divergent* if the run is infinite and the sum over all time steps $\sum_{i\geq 0} t_i$ is also infinite. Moreover, a hybrid automaton $\mathcal{H}$ is called *non-Zeno* if there is no time divergent run containing infinitely many discrete steps [ACH+95].

Figure 2.2 illustrates an example of a hybrid automaton modelling a simple car.

Example 2.1.1 (Hybrid Automaton)**.** *Consider the situation illustrated in Figure 2.1. The left car should be modelled by a hybrid automaton. For this purpose it is assumed the the car driving ahead (the right one) drives with a constant velocity* v_c *and the distance* d *between the two cars is larger then a critical distance* $\mathrm{d_c}$. *Figure 2.2*

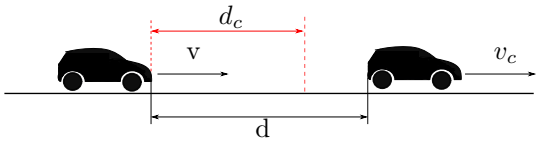


Figure 2.1: Situation modelled by the hybrid automaton illustrated in Figure 2.2.

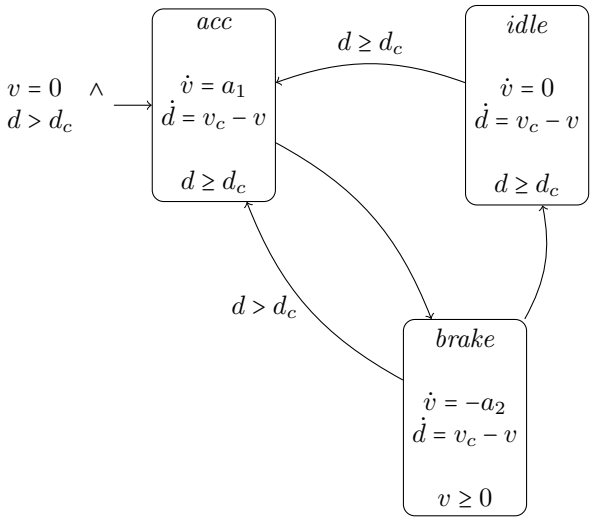


Figure 2.2: Hybrid automaton modeling a simplified version of a car.

shows the corresponding hybrid automaton. The left car can either accelerate, brake or do nothing. Therefore, the hybrid automaton has the three locations `acc`, `brake`, *and* `idle` *respectively. The quantities of interest are the velocity* v *of the car and the distance* d *between the car and the car driving ahead.*

Initially the car is accelerating. This is depicted by an arrow pointing to the location `acc` *that has no predecessor location. The initial value of the velocity is zero and the distance to the next car is greater than the critical distance* $\mathrm{d_c}$*. During the acceleration the car changes its velocity according to some constant acceleration* $\mathrm{a_1}$*. The distance change corresponds to the difference between the velocity of the left and right car. Therefore,* $\dot{\mathrm{d}}$ *is set to* $\mathrm{v_c} - \mathrm{v}$*. The car can accelerate only as long as the distance is greater than* $\mathrm{d_c}$*. Thus, the invariant* $\mathrm{d} \geq \mathrm{d_c}$ *is needed. Once the invariant is violated the car has to brake. Since braking should be allowed any time, the transition from* `acc` *to* `brake` *has no guard. While braking the velocity decreases with the deceleration* $\mathrm{a_2}$*. The distance change remains the same as in* `acc`*. The control is allowed to stay in the location* `brake` *as long as the velocity is greater than zero. The location has two outgoing transitions, one with the target location* `acc` *and the other with the target location* `idle`*. On the one hand, after braking it is again allowed to accelerate but if and only if the distance* d *is greater than* $\mathrm{d_c}$*. On the other hand, if the car neither brakes nor accelerates it switches to the* `idle` *state. In this location only the distance changes. In the real world, the velocity would of course change due to, e.g., fraction, but here it is neglected. The control is allowed to stay in* `idle` *as long as the distance* d *is greater than* $\mathrm{d_c}$*. The same is required by the guard of the outgoing transition with*

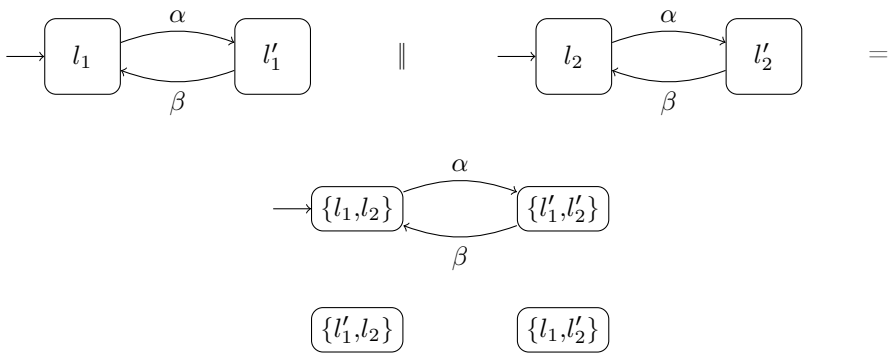


Figure 2.3: Example of parallel composition of two hybrid automata.

the target location `acc`.

Due to the fact that many systems are compositional, the parallel composition of hybrid automata is introduced now.

Definition 2.1.3. *Parallel Composition of Hybrid Automata ([Hen00]) Consider two hybrid automata* $\mathcal{H}_1 = (Loc_1, Var, Lab_1, Trans_1, Flow_1, Inv_1, Init_1)$ *and* $\mathcal{H}_2 = (Loc_2, Var, Lab_2, Trans_2, Flow_2, Inv_2, Init_2)$. *Their* parallel composition $\mathcal{H}_1 \| \mathcal{H}_2$ *is a hybrid automaton* $\mathcal{H} = (Loc, Var, Lab, Trans, Flow, Inv, Init)$ *with*

- $Loc = Loc_1 \times Loc_2$,
- $Lab = Lab_1 \cup Lab_2$,
- $((l_1,l_2),\alpha,\mu,(l'_1,l'_2)) \in Trans$ *iff*
  - $(l_1,\alpha_1,\mu_1,l'_1) \in Trans_1$ *and* $(l_2,\alpha_2,\mu_2,l'_2) \in Trans_2$, *and*
  - *either* $\alpha_1 = \alpha_2 = \alpha$ *or* $\alpha_1 = \alpha \notin Trans_2$ *and* $\alpha_2 = \tau$, *or* $\alpha_1 = \tau$ *and* $\alpha_2 = \alpha \notin Lab_1$, *where* τ *is the stutter label, and*
  - $\mu = \mu_1 \cap \mu_2$
- $Flow(l_1,l_2) = Flow_1(l_1) \cap Flow_2(l_2)$ *for all* $(l_1,l_2) \in Loc$,
- $Inv(l_1,l_2) = Inv_1(l_1) \cap Inv_2(l_2)$ *for all* $(l_1,l_2) \in Loc$,
- $Init = \{((l_1,l_2),\nu) \mid (l_1,\nu) \in Init_1, (l_2,\nu) \in Init_2\}$

An example of parallel composition of two hybrid automata is depicted in Figure 2.3.

2.1.1 Linear Hybrid Automata

There exist a number of different types of hybrid automata, that differ in the type of dynamics and the type of the expressions defining guards and resets. The simplest type of hybrid automata are *timed* automata [HKPV98]. Their flows are defined by *clocks* which are real-valued variables evolving with the constant slope of one and its values can only be reset to zero. A more expressive class of hybrid automata are *rectangular* automata [HKPV98] whose flows are defined by intervals. The focus of

this thesis lies on the super-class of rectangular automata - the *linear hybrid automata* (LHA) whose dynamics is defined by linear ordinary differential equations. Sometimes, one distinguishes between LHA I and LHA II. The dynamics of LHA I is defined by constants while the dynamics of LHA II are defined by linear ODEs. This thesis focuses on LHA II and for simplification they will be abbreviated with LHA [HKPV98].

Depending on how the flow of the linear hybrid automaton is specified, one can distinguish between *autonomous* and *non-autonomous* hybrid system. The flow of an autonomous LHA system is characterized by a system of linear ODEs of the following form

$$\dot{x}(t) = A \cdot x(t), \text{ where } A \in \mathbb{R}^{n \times n}. \tag{2.3}$$

The flow of a non-autonomous system additionally considers external input, i.e. disturbance caused by the environment. For this purpose the Equation 2.3 is extended by a time-dependent function *u(t)*, the dynamics can be specified by the following system

$$\dot{x} = A \cdot x + B \cdot u(t), \text{ where } A \in \mathbb{R}^{n \times n} \text{ and } B \in \mathbb{R}^{n \times m}. \tag{2.4}$$

In order to compute the *flowpipe*, which is the topic of next section, the solution of the Equation 2.3 is needed, i.e. a vector $x(t)$ is sought which satisfies the initial condition

$$x(0) = x_0. \tag{2.5}$$

The combination of Equations 2.3 and 2.5 is called *initial value problem* (IVP) and its solution is given by the following equation [ASB07]:

$$x(t) = e^{tA} \cdot x_0. \tag{2.6}$$

The expression $e^{t \cdot A}$ is called *matrix exponential* and can be approximated by the following power series [ASB07]:

$$e^{tA} = \sum_{k=0}^{\infty} \frac{(tA)^k}{k!} = \underbrace{\frac{(tA)^0}{0!}}_{=Id} + \frac{(tA)^1}{1!} + \frac{(tA)^2}{2!} + \cdots. \tag{2.7}$$

2.2 Reachability Analysis of Hybrid Systems

In order to verify if certain properties are satisfied by a hybrid automaton, one has to explore the reachable states of the hybrid automaton. This approach is called reachability analysis and is discussed in this section in detail.

2.2.1 The Reachability Problem

In practice systems are often accompanied with specifications of properties that the systems must fulfill. The question if a system actually fulfills the specification, is analog to the reachability problem that concerns the question if a system can reach a certain state, or a set of states, starting at a defined initial state.

Usually, two types of properties are considered: *safety* and *liveness* properties. Informally speaking, safety properties are properties stating that "nothing bad will happen". In terms of the car example, the property that the distance between two successive cars is always greater than some critical distance, is an example of a safety property. On the other hand liveness properties describe that "always something good

will happen". An example is the property stating that whenever a car drives, it will always stop at some point in time [BK08].

One possible way to verify if a system under consideration fulfills given properties, is to perform the *reachability analysis* of the hybrid automaton that models the system. The basic idea of this approach is the computation of reachable sets of states and the examination if the sets of states intersect with *bad sates*, the states that do not fulfill the properties of interest [Ábr12].

Unfortunately, the reachability problem is not decidable for all types of hybrid systems. In [HKPV98] the authors show that reachability problem is only decidable for few types of hybrid automata. Prominent examples are timed automata and initialized rectangular automata [Ábr12]. Any relaxations concerning the initialization of the automaton, result in undecidability of the system. Regrettably, the classes of hybrid automata for which the reachability problem is decidable, are usually not expressive enough. The reachability problem for linear hybrid automata is undecidable. Fortunately, the *bounded* reachability, i.e., reachability within a predefined number of steps and amount of time, is still decidable when the sets of states are over-approximated [Ábr12].

2.2.2 Reachability Analysis of LHA

Over the past years, many different tools computing the reachable states of a hybrid automata have been developed. Some popular examples are UPPAAL [LPY97], SPACEEX [FLGD+11], CORA [AKA18], and FLOW* [CÁS13]. All the tools compute the reachable states based on the flowpipe construction. A single flowpipe corresponds to the states that are reachable within a single location within an amount of time. Note that the number of reachable states is infinite, as the states variables are continuous. Therefore, the reachability analysis algorithm is based on the successive computation of finite sets of states.

In order to guarantee the termination of the reachability analysis algorithm, one can limit the time that should be spent on computing the reachable states (called *time horizon*) as well as the number of discrete jumps (called *jump depth*). Since LHA have non-linear behaviour, usually one computes an *over-approximation* of the reachable states, in order to capture the overall dynamics. Note that the over-approximation has an impact on the safety verification. If the over-approximated set of reachable states does not violate the safety specification, then it can be safely assumed that the precisely computed reachable states neither violate them. However, if the over-approximation violates the specifications, one cannot guarantee that the precise flowpipe also violates them [Ábr12].

This thesis focuses on the *forward reachability analysis* which basic idea is depicted by Algorithm 1.

The set R collects all reachable sets of states while the set R_{new} collects new sets of reachable states within the next computation step. Initially, both sets are equal to the set of initial states. The while-loop is only entered, if there are still some new states to process and the termination condition *termination_cond* is not satisfied. The termination condition is needed in order to guarantee the termination of the algorithm. The condition can test if the jump depth or time horizon was already reached but also detect if the current new states were already explored (i.e., if a fixed-point was reached) [Ábr12].

In the while-loop each of the new state sets is processed individually. Once one

Algorithm 1 Flowpipe Construction Based Forward Reachability Analysis.

Input: Hybrid automaton $\mathcal{H}$
Output: Set of reachable states R

$R := Init_{\mathcal{H}}$
$R_{new} := R$ {the termination_condition is needed to guarantee the termination}
while $R_{new} \neq \emptyset \wedge \neg$termination_cond **do**
  let $stateset \in R_{new}$
  $R_{new} := R_{new} \setminus \{stateset\}$
  $R' :=$ computeFlowPipe($stateset$)
  $R'' :=$ computeJumpSuccessors(R')
  $R_{new} = R_{new} \cup (R'' \setminus R_{new})$
  $R = R \cup R' \cup R''$
end while
return R

state is non-deterministically selected, the flowpipe for the state can be computed, i.e. all reachable states within the time horizon. Those states are stored in the set R'. Next, the jump successors are computed by the function `computeJumpSuccessors`. This function tests if the reachable states contained in the flowpipe intersect any guards, and if so then the discrete jump successor states are added to the sets R_{new} and R.

A closer look at the flowpipe construction needs to be taken, as its computation is not trivial. As already mentioned a flowpipe is a set $\Omega_0, \ldots, \Omega_n$ of state sets that are reachable within a defined amount of time. Each of the state sets Ω_i represents the set of reachable states within a time interval of size δ, called *time step*. There exist a number of different state set representations, some of them will be discussed in the next section. For autonomous systems (see Equation 2.3) the set of states reachable from a state Ω_i at time δ can be computed as follows

$$\Omega_{i+1} = e^{\delta A} \cdot \Omega_i \tag{2.8}$$

where $e^{\delta A}$ is the matrix exponential defined in Section 2.1.1. Since the systems considered here exhibit non-linear behaviour, one needs to over-approximate the sets in order to cover the overall dynamics of the system. This over-approximation is called *bloating*. Bloating means that the convex hull of the states is widened by a factor. For autonomous systems it is sufficient to bloat only the initial sets by some factor α. In case of non-autonomous systems, an additional bloating by factor β has to be added in order to capture the influence of external input. Therefore, the set of states reached during the first time interval can be computed by

$$\Omega_0 = conv(X_0 \cup e^{\delta A} X_0) \oplus \mathcal{B}_{\alpha+\beta} \tag{2.9}$$

where X_0 is the initial state and $\mathcal{B}$ is the bloating. Figure 2.4 illustrates the method [FLGD+11].

Figure 2.5 illustrates an exemplary reachability analysis. The figure presents the reachable states of two locations l_1 and l_2. The red quadrangles depict bad states, the white ones are invariants, and the gray ones are the guards. The dark blue square, is the bloated initial state. For the first flowpipe, the actually reachable sets are

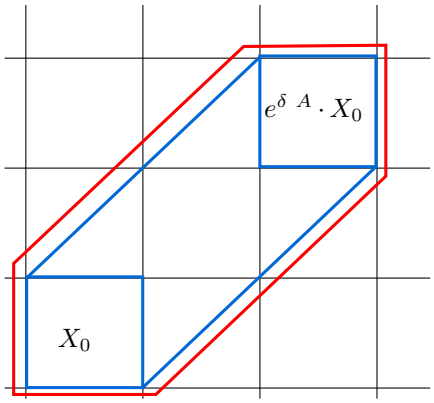


Figure 2.4: Example of bloating the first time interval.

illustrated by the two smooth lines and the small square on the blue initial state. The jumps are depicted by arrows.

2.3 State Set Representations and Set Operations

In order to implement the reachability analysis algorithm (Algorithm 1), first one needs to implement the representations for the states reachable by a hybrid automaton as well as a set of operations that have to be performed during the analysis. Since the precision and performance of the reachability analysis strongly depends on the state set representation, usually a set of different state set representations are implemented. Therefore, subsequently several state set representations are introduced as well as the operations that are needed for the analysis.

2.3.1 State Set Representation

The state space of hybrid automata is infinite since they deal with continuous variables [Hen00]. Therefore, a simple enumeration of reachable states is not feasible. Instead, the set of reachable states is computed with finite *symbolic* representations of the infinite regions [Hen00]. For example the set x of all real numbers between 0 and 1 is infinite, the finite symbolic representation of this set is the logical formula $0 \leq x \wedge x \leq 1$ which is basically an interval. Established symbolic representations for state set of hybrid systems are *support functions* or *Taylor models* [Hen00].

Another possibility to represent sets of states are geometric representations. For this purpose usually *convex sets* are used. The most famous representations are *boxes*, *polyhedra*, *zonotopes*, *ellipsoids*, etc. Some of the representations will be discussed later in this section. Note that the sets considered here are convex as it is easier to guarantee the conditions required by the semantics of hybrid automata (see Definition 2.1.2). The crucial point is that the invariant does not only need to be satisfied at the beginning and the end of the computation but also in between the two points in time. When representing sets by convex sets, one can guarantee that the invariant is satisfied at any point in time if it is satisfied at the beginning and end of the computation.

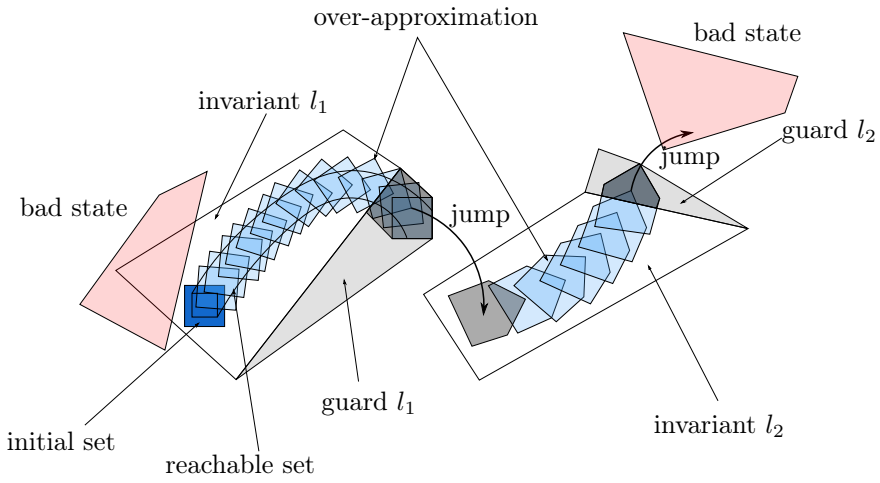


Figure 2.5: Sketch of a reachability analysis.

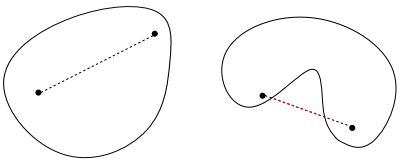


Figure 2.6: Difference between convex (left) and not convex set (right).

This however, does not hold for non-convex sets and would make the computations more complex. This phenomenon is illustrated in Figure 2.6.
Formally, convex sets are defined as follows:

Definition 2.3.1. ***Convex Set*** *[Zie14] A set S is called* convex*, if*

$$\forall x, y \in S. \forall \lambda \in [0,1] \subseteq \mathbb{R}. \lambda \cdot x + (1 - \lambda) \cdot y \in S.$$

There exists a large number of various geometric representations for state set of hybrid systems, as each of the representations has different advantages and disadvantages. Choosing a representation is always a struggle between computational effort, precision of the resulting representation and the amount of storage needed to save the representation. Usually, it holds that the more precise the representation, the more computational effort is needed. Depending on which operations (intersection, union, Minkowski sum, linear and affine transformations, etc.) should be performed on the sets, the efficiency of the computation varies for the different representations [Ábr12, Hen00]. This will be discussed later on but now some representations will be introduced.

Boxes. A box can be represented by a sequence of intervals $(I_0, \ldots, I_d)$ or by its maximal and minimal points. Both representations are depicted in Figure 2.7a.

Definition 2.3.2. ***Boxes*** *[MKC09] A set $\mathcal{B} \subseteq \mathbb{R}^d$ is a* box *if there exist intervals $I_0, \ldots, I_d \in \mathbb{I}$ such that*

$$\mathcal{B} = I_0 \times \cdots \times I_d.$$

Polytopes and Polyhedra. Another representation are convex *polytopes* and *polyhedra*. In order to define them, first the definition of closed halfspace is needed.

Definition 2.3.3. ***Closed Halfspace*** *[Zie14] A d-dimensional* closed halfspace *is a set $\mathcal{H} = \{x \in \mathbb{R}^d \mid c \cdot x \leq z\}$ for some $c \in \mathbb{R}^d$, called the* normal *of the halfspace, and a offset $z \in \mathbb{R}$.*

Convex polyhedra are defined as an intersection of a finite set of *halfspaces*. A halfspace can be defined by a normal vector and an offset as stated in the Definition 2.3.3.

Definition 2.3.4. ***Convex Polyhedron*** *[Zie14] A set $\mathcal{P} \subseteq \mathbb{R}^d$ is a* convex polyhedron *if there are $n \in \mathbb{N}$ and $c_i \in \mathbb{R}^n, d_i \in \mathbb{R}, i = 1, \ldots, n$ such that*

$$\mathcal{P} = \bigcap_{i=1}^{n} h_i, \text{ where } h_i = \left\{ x \in \mathbb{R}^d \,\middle|\, c_i \cdot x \leq d_i \right\}.$$

If the resulting polyhedron is closed, then it is called a *polytope*. A polytope can alternatively be defined by a set of vertices. In order to obtain a polytope from a set of vertices, one has to compute the *convex hull* which is defined as follows:

Definition 2.3.5. ***Convex Hull*** *[Zie14] Given a set $V \subseteq \mathbb{R}^d$, the* convex hull *conv(V) of V is the smallest convex set that contains V. For a finite set $V = \{v_1, \ldots, v_n\}, n \in \mathbb{N}$ the convex hull can be computed by*

$$conv(V) = \left\{ x \in \mathbb{R}^d \,\middle|\, \exists \lambda_1, \ldots, \lambda_n \in [0,1] \subseteq \mathbb{R}^d . \sum_{i=1}^{n} \lambda_i = 1 \wedge \sum_{i=1}^{n} \lambda_i \cdot v_i = x \right\}$$

A polytope defined by the intersection of a finite set of halfspaces is called $\mathcal{H}$-representation, while the representation as a set of vertices is called $\mathcal{V}$-representation.

Definition 2.3.6. ***$\mathcal{H}$-Polytope, $\mathcal{H}$-Polyhedron*** *[Zie14] A d-dimensional $\mathcal{H}$-*polyhedron *$P = \bigcap_{i=1}^{n} \mathcal{H}_i$ is the intersection of finitely many closed halfspaces. A bounded $\mathcal{H}$-*polyhedron *is called an $\mathcal{H}$-polytope.*

An example is shown in Figure 2.7b.

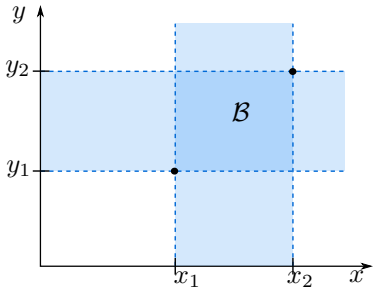
Support Functions. Support functions belong to the class of symbolic state representations. They are an important representation, as most operations can be computed quite efficient on them.

Definition 2.3.7. ***Support Function*** *[LGG09] A* support function *is a function $\sigma : \mathbb{R}^d \to \mathbb{R}$ defining a set*

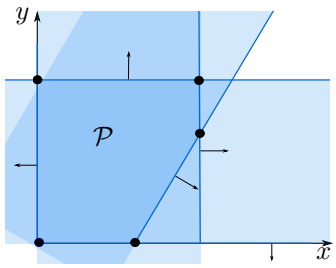
$$\mathcal{S} = \left\{ x \in \mathbb{R}^d \,\middle|\, r \cdot x \leq \sigma(r) \forall r \in \mathbb{R}^d \right\},$$

where $\sigma(r) \in \mathbb{R}$ is called the support value for the given direction $r \in \mathbb{R}^d$.

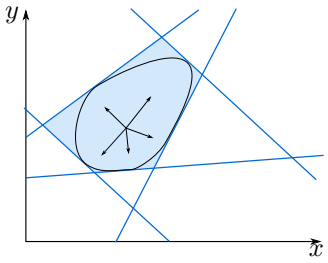
An example is illustrated in Figure 2.7c.



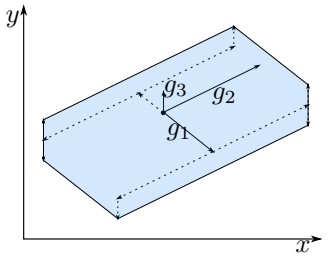
(a) Example of a set represented by a box.



(b) Example of of a set represented by a polytope.



(c) Example of a set represented by a support function.



(d) Example of a set represented by a zonotope with three generators.

Figure 2.7: Examples of three different state sets representations.

Zonotopes. Zonotopes are point-symmetric sets that can be defined very compactly by a center point and a set of line segments, called *generators*. The set is then computed by the Minkowski sum of the generators shifted to the center. The formal definition is as follows.

Definition 2.3.8. ***Zonotope*** *[Gir05] A set $\mathcal{Z} \subseteq \mathbb{R}$ is a* zonotope *if there is a center $c \in \mathbb{R}^d$ and a finite set $G = \{g_1, \ldots, g_n\}$ of* generators *$g_i \in \mathbb{R}^d$ such that*

$$\mathcal{Z} = \left\{ x \,\middle|\, x = c + \sum_{i=1}^{n} \lambda_i \cdot g_i, -1 \leq \lambda_i \leq 1 \right\}.$$

An example is depicted in Figure 2.7d. An important property of a zonotope is the *zonotope order*. The order of a d-dimensional zonotope described by n generators is defined as $ord(\mathcal{Z}) = \frac{n}{d}$.

2.3.2 Operations on State Sets

In order to compute the set of reachable states several set operations are needed. These include the convex hull of union, intersection, Minkowski sum, affine transformation, and the tests for emptiness and membership. Let D be a domain, and A,B,$S \subseteq D$.

- $\cdot \bigcup \cdot$ (*union*): The union of two state set representations is defined as

  $$A \bigcup B = conv\{x \mid x \in A \text{ or } x \in B\}.$$

  The operation *conv* is the convex hull operation, which is needed here, since convex state set representations are not closed under the operation union (the resulting set might be non-convex). This operation is needed for the computation of the first segment of a flowpipe and whenever aggregation is used.

- $\cdot \bigcap \cdot$ (*intersection*): The intersection operation is defined as

  $$A \bigcap B = \{x \mid x \in A \text{ and } x \in B\}.$$

  The intersection operation is a frequently used operation during the flowpipe construction, as it is needed for testing the satisfiability of invariants, intersection with guards and bad states, and the detection of fixed-point.

- $\cdot \oplus \cdot$ (*Minkowski sum*): The Minkowski sum is defined as

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

  This operations is needed whenever bloating should be performed.

- $A(\cdot)$ (*affine transformation*): The operation is defined as

  $$A(S) = \left\{ x \,\middle|\, x = A \cdot y + b, y \in S, A \in \mathbb{R}^{d \times d}, b \in \mathbb{R}^n \right\}.$$

  The affine transformation is another frequently used operation needed whenever the next flowpipe segment from the current one should be computed.

- *Test for emptiness* is a predicate which is satisfied if $S = \varnothing$ holds.

- *Test for membership* is a predicate that is satisfied if $x \in S$ for $x \in D$.

The performance of the operations depends on the chosen set representations. Table 2.1 summarizes the complexities of the operations for the different state set representations. A plus means that the operation can be performed in polynomial time, while a minus means that the operation needs exponential time. According to this table, the support functions and $\mathcal{V}$-polytopes can perform good on three out of four operations. However, one has to keep in mind that during the reachability analysis the operations are used in different frequencies. For example for autonomous systems Minkowski sum and union are only used once to compute the first segment. For non-autonomous systems Minkowski sum and union are frequently used operations.

Set Repr.	$\cdot \bigcup \cdot$	$\cdot \bigcap \cdot$	$\cdot \oplus \cdot$	$A(\cdot)$
Box			+	
$\mathcal{H}$-Polytope	-	+	-	-
$\mathcal{V}$-Polytope	+	-	+	+
Support Function	+	-	+	+
Zonotope			+	+

Table 2.1: Complexities of the operations for different state set representations. This table is taken from [Ábr12].

Chapter 3

Comparison of CORA and HyPro

In this chapter two tools, CORA and HyPro, for reachability analysis of hybrid systems are compared with each other. Since CORA is a toolbox for MATLAB, first the usability of hybrid systems in MATLAB will be motivated. Subsequently, the two tools will be introduced. Finally, the wrapper for HyPro, MHyPro, will be presented.

3.1 MATLAB and Hybrid Systems

MATLAB (MATrix LABoratory) is a widely used tool among scientists and engineers for the development and analysis of systems. MATLAB was originally developed for matrix computations, therefore, the programming language is matrix-based. Today MATLAB includes far more functionalities than matrix manipulation functions. It provides a huge number of algorithms and toolboxes for various areas of applications such as machine learning, signal processing, image processing, computer vision, etc. It has a build-in plotting engine so it is easy to visualize data and results. Moreover, MATLAB offers interfaces to other programming languages such as, C/C++, Java, Python, and Fortan [Mat].

Since MATLAB provides toolboxes like SIMULINK and STATEFLOW, it is used for the development of CPSs, as model-based design is an essential step in the development processes of CPSs. Due to the fact that CPSs are mostly safety-critical and require validation tests, hybrid systems theories were developed in order to support the process of validation [Lee10]. This circumstances motivate the integration of verification methods for hybrid systems in MATLAB [DLV11].

There is already one toolbox for simulation of hybrid systems for MATLAB called *Hybrid Equations* (HyEQ) *Toolbox* [SCN13]. However, simulations cannot explore all reachable states, therefore, a toolbox for reachability analysis is needed. A prominent toolbox that implements reachability algorithm for hybrid systems is CORA [AKA18] which will be introduced in the next section.

3.2 CORA

The Continuous Reachability Analyzer (CORA) is a prominent tool implemented in MATLAB providing reachability algorithms for systems with continuous, discrete, and hybrid dynamics. The idea of CORA is the possibility of implementing customized reachability algorithms in relatively short time. This is especially possible because CORA is an object-oriented toolbox that uses modularity, operator overloading, inheritance, and information hiding, hence, the user can easily replace a module by a more tailored one [AKA18].

3.2.1 Modelling Systems in CORA

The basis of all reachability analysis tools are the state set representations. As mentioned in Section 2.3.2, for reachability analysis the choice of state set representation is crucial, as the efficiency of the operations differs for each representation. Besides popular set representations, like intervals, zonotopes, Taylor models and vertices, CORA introduces some more special representations. Examples are the *zonotope bundles*, that will be shortly discussed later on, and *polynomial, probabilistic,* and *constraint zonotopes*. Polynomial zonotopes have been introduced in [Alt13] and are needed, as zonotopes are not closed under nonlinear maps. In order to enable stochastic verification, probabilistic zonotopes are implemented that have been introduced in [ASB09]. The constrained zonotopes [SRMB16], are zonotopes with additional constraints on the generators that define the corresponding zonotope [AKA18].

CORA implements reachability and simulation algorithms for a number of different classes of continuous dynamics which can also be used to describe the dynamics of hybrid systems. Linear dynamics is the most basic supported dynamics. Additionally, one can add uncertain, fixed or varying parameters to the system. Moreover, CORA supports linear probabilistic systems, where the dynamics is defined by a set of linear stochastic differential equations [Gar85]. Besides linear dynamics, CORA also supports non-linear systems, discrete-time non-linear systems, non-linear systems with uncertain fixed parameters, and non-linear differential-algebraic systems [AKA18].

The palette of supported dynamics makes CORA attractive for many areas of applications. There is a number of systems that for example require the support of differential-algebraic systems. They provide so called *algebraic variables* that are not state variables and can be introduced as an invariant that expresses physical laws like for example Newton's laws. This type of variables is not supported by many tools, SPACEEX is an exception [DF13].

There are three ways to model systems in CORA. The first option is the direct implementation using MATLAB language. The second option is the usage of SPACEEX syntax, as CORA provides a converter that can convert SPACEEX models to CORA models. Since SPACEEX provides a simple GUI for the development of hybrid automata, it is an important step towards user-friendly usability. The last alternative is SIMULINK. CORA offers a converter that can convert SIMULINK models into SPACEEX which can be subsequently converted to CORA [AKA18].

Since many systems are too complex to model them by a single hybrid automaton, it is common to model each subsystem by a separate hybrid automaton. Subsequently the parallel composition of the automata is computed. The problem about the parallel composition of hybrid automata, is the fact that the number of locations of the resulting hybrid automaton increases exponentially. However, not all of the computed

locations are reachable. Therefore, besides the classical hybrid automaton, CORA also provides a parallel hybrid automaton, that is a data structure for parallel composition of several hybrid automata. The system is then described by a list of several hybrid automata and the reachable sets are computed on-demand, i.e., only if the control reaches the corresponding sets.

3.2.2 Parameters

As every reachability analysis tool, CORA requires a set of options that need to be determined by the user. The correct choice of the parameters is crucial for the analysis results.

Besides standard parameters as initial set, initial and final location, time-step size, and time horizon, CORA requires some more specific options. Depending on the chosen dynamics the parameters which have to be determined vary. Since in this thesis only linear hybrid systems are considered, only the parameters needed for this class of systems will be considered and shortly explained.

- *Time horizon* - Since for linear hybrid systems only bounded reachability is decidable, CORA requires that a maximal time amount for the overall analysis is determined. The larger the time horizon, the more reachable states can be explored, and therefore also the time needed for the computations gets higher.
- *Time-step size* - Each flowpipe consists of segments. The size of a segment is defined as the time-step size. The larger the time-step size, the bigger the segments and the more imprecise the flowpipe. However, the lower the time-step size, the more computations have to be performed and therefore the more time is needed for the computation of the flowpipe.
- *Reduction technique for the zonotope order* - The performance of the set operations on zonotopes depends on the order of the zonotope. The higher the zonotope order, the more time is needed to perform the required set operations. Hence, CORA implements several methods for the order reduction of zonotopes that have been introduced in [KSA17]. The methods yield different over-approximations and perform differently. The default technique is `girard` which has been introduced in [GH04].
- *Taylor terms* - The matrix exponential needed for the computation of flowpipes, can be over-approximated by the Taylor series. This parameter determines how many Taylor terms of the series will be considered for the computation. The more Taylor terms are considered, the tighter the resulting over-approximation of the remainder term of the Taylor series [Alt10]. Thus, the precision of the flowpipe increases with the increasing number of considered Taylor terms. However, the computation time also increases.
- *Zonotope order* - This parameter defines the maximum order of the zonotopes. A high zonotope order, means a high number of generators. The more generators are used for the definition of a zonotope, the more precise the set can be represented. However, the computation time increases with growing zonotope order, since the performance of the operations sinks with the growing number of generators.
- *Guard intersection method* - CORA offers two approximation methods for the intersection with potential guard sets - the `polytope` and the `zonoGirard`

methods. Both methods yield a set of enclosed zonotopes for the intersection with potential guard sets. The enclosure method is determined by the parameter that will be introduced next. The choice of the method also determines the representation of the guards sets. When `polytope` is the chosen method, then the guards have to be represented by polytopes, that can be defined by a set of halfspaces. Otherwise, when using the `zonoGirard` method, the guards have to be represented by a single halfspace.

- *Enclosure method* - CORA provides five different enclosure methods for polytopes. The result is an over-approximating zonotope that encloses the polytopes. Each method uses a different zonotope for the over-approximation.

- *External input* - CORA provides reachability algorithms that consider external input. The external input may be constant or uncertain and can be individually set for each location.

These parameters are only a subset of all parameters that can be set in CORA. On the one hand, the high number of parameters makes it possible for the user to customize the models very precisely. On the other hand, the correct choice of the parameters is not easy and is crucial for the results of the reachability analysis. Wrong choice of the parameters can result in very imprecise computations or even make the reachability analysis impossible. Due to the fact that most of the parameters require expert knowledge, the access to the tool for non-experts is difficult. The developers of CORA are aware of this problem and plan to introduce self-tuning of the parameters in the future.

3.2.3 Reachability Analysis

Although CORA provides a palette of state set representations, the reachability analysis is performed on zonotopes. Zonotopes perform good for linear transformation and Minkowski sum [AK11]. Hence, this is a reasonable choice especially for non-autonomous systems where the Minkowski sum needs to be performed when ever a successor segment should be computed. This advantage, however, comes with a high price as zonotopes suffer from the fact that they are not closed under intersection. Since the intersection operation is a constantly used operation during the reachability analysis of hybrid systems, CORA introduces the zonotope bundles mentioned earlier [AK11]. The idea of zonotope bundles is that the zonotopes that should be tested for intersection with a set, are stored in a list. The exact intersection is not computed, instead every zonotope in the bundle is over-approximated by a polytope and the intersection with guards and invariants is performed for each polytope separately. The computational costs is the cost of the operation for a single zonotope times the number of zonotopes in the bundle [AK11].

Other operations can also be performed on the bundles. Consider the Minkowski sum of a zonotope bundle with a single zonotope

Definition 3.2.1 (**Minkowski Sum for Zonotope Bundles** [AK11])**.** *The Minkowski addition of a zonotope bundle $\mathcal{Z}^{\cap}$ and a zonotope $\mathcal{Z}^{add}$ is over-approximated by*

$$\mathcal{Z}^{\cap} \oplus \mathcal{Z}^{add} \subseteq \left\{ \mathcal{Z}_1 \oplus \mathcal{Z}^{add}, \ldots, \mathcal{Z}_n \oplus \mathcal{Z}^{add} \right\}^{\cap}.$$

The overall process of the computation of reachable state sets is as showed in Algorithm 1. The termination condition only concerns the time horizon, the jump depth is ignored. However, it would be easy for a user to customize the algorithm such that it considers the jump depth. This is reasonable, as the jump depth is only useful in theoretical context and does not have any meaning in the real world. However, it would be easy for a user to customize the algorithm such that it considers the jump depth. This is especially possible because the parameters are stored as a struct, so new options can be added straight forward and it does not require any changes in the implementation.

3.3 HyPro

HyPro is an open-source C++ library which provides implementations of the most popular state set representations. The basic idea of HyPro is the introduction of various state set representations in order to offer assistance for the implementation of customized algorithms [SÁMK17].

3.3.1 Modelling Systems in HyPro

There are two possible ways to model systems in HyPro. First possibility is to implement the models directly using C++ which is a pretty involving process. A more intuitive and easier way to model systems is the usage of Flow* syntax, as HyPro provides a parser for the syntax [SÁMK17].

HyPro supports systems exhibiting linear behaviour which is a restriction. However, a lot of non-linear systems can be linearized. Similarly as Cora, HyPro also provides a number of set representations. Those include boxes, polytopes, halfspaces, zonotopes, support functions, orthogonal polyhedra, and Taylor models. For each of the representation, except for orthogonal polyhedra and Taylor models, HyPro offers a set of functions that are needed to perform reachability analysis, for example union, intersection, Minkowski sum, etc. Moreover, unlike in Cora, in HyPro each of the set representations can be used as set representation for the reachability analysis. Thus, the user has to decide if for example boxes, that usually produce big over-approximations, are suitable for a system, or if it is better to use for example support functions that usually produce tighter over-approximations than boxes but need more computation time. Another feature for the state set representations is the fact that every representation is templated in number, hence, HyPro can make use of the `boost` library for exact arithmetic computations [SÁMK17].

As already mentioned in the last section, parallel composition is a useful method to model complex system more easily. Unfortunately, HyPro does not provide the possibility of modelling systems as parallel composition of multiple hybrid automata. However, the developers currently work on the implementation of this functionality.

3.3.2 Parameters

Similarly as Cora, HyPro also requires the determination of parameters. Both tools have one common parameter - the time-step size. The remaining parameters required by HyPro are shortly explained now.

- *Time horizon* - This parameter determines how long the computation of a single flowpipe should be. Thus, the time horizon in HYPRO is related to a single flowpipe. In CORA the time horizon is related to the overall analysis.
- *Jump depth* - The jump depth defines how many times the control may take a discrete transition. Thus, the jump depth limits the number of computed flowpipes.
- *Clustering* - Whenever a flowpipe intersects with a guard set, the successor flowpipes need to be computed. Each of the flowpipes segments that are inside the guard set are potential initial sets for a successor flowpipe. Since the number of segments might be big, the computation might get very slow. In order to accelerate the process, one can cluster several segments into a single segment that subsequently will be used as the initial set for the next flowpipe. Indirectly, the parameter determines the maximal number of successor flowpipes after an intersection with a guard set. If the parameter value is set to zero then the algorithm uses aggregation, i.e., all segments intersecting with a guard set are over-approximated by a single segment.
- *Bloating type* - In order to capture all trajectories, the initial set needs to be over-approximated, this is called bloating. One can choose between uniform and non-uniform bloating. If the parameter is not set then the non-uniform bloating is used.
- *Set representation* - This parameter determines which state set representation is used for the reachability analysis.

3.4 MHYPRO

Due to the fact that a number of CPSs and other systems are developed in SIMULINK and MATLAB, it is sensible to implement the verification tools directly in MATLAB. Therefore, a MATLAB-wrapper for HYPRO- MHYPRO- was developed in context of this thesis.

MHYPRO consists of a collection of MATLAB classes which are wrappers for the corresponding classes in HYPRO. It offers three different state set representations - boxes, support functions, and zonotopes. Moreover, it wraps the most important data structures for hybrid automata, i.e., locations, transitions, conditions (needed for the formulation of invariants and guards), flows, labels, and hybrid automata. For the reachability analysis, it also wraps the classes `Reach` and `State`. Figure 3.2 illustrates the general structure of MHYPRO.

The implementation of the wrapper is based on the MATLAB C MEX API [Mat] that introduces methods needed for usage of C/C++ code in MATLAB. The most important function of the API is the so called *MEX function* that serves here as the gateway between MATLAB and HYPRO. Thus, every time MHYPRO calls a function, first it calls the MEX function that can process the passed arguments and call the correct function contained in the C++ component of MHYPRO which then can call the desired function in HYPRO. The class handler is needed to guarantee correct communication between MATLAB and HYPRO. Its purpose is the correct conversion of pointers from MATLAB to HYPRO and vice verse. The object handler is needed for the conversion of the data structures. For example the handler can convert HYPRO

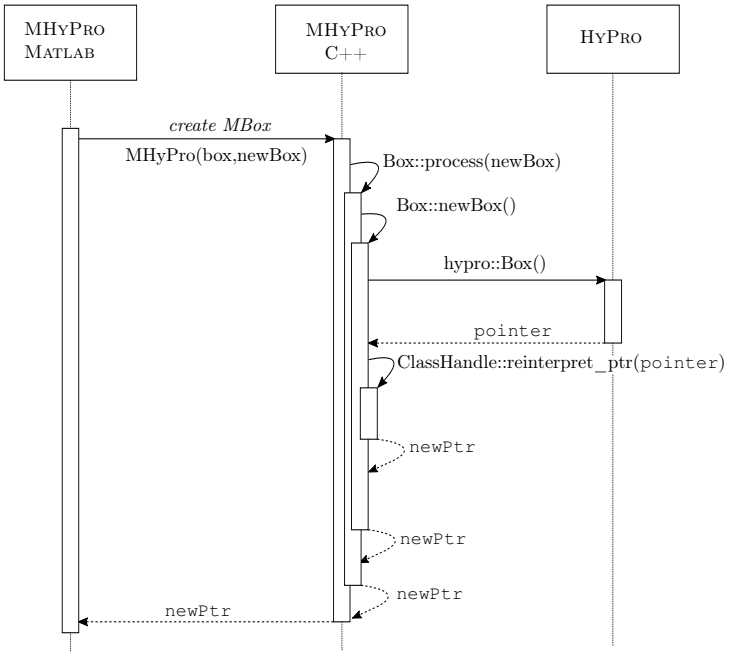


Figure 3.1: Sequence diagram showing the process of creating a new box in MHyPro.

matrices into Matlab matrices and vice verse. Since Matlab provides plotters for various geometric objects, two plotters, a 2D and a 3D plotter for the flowpipes were implemented.

Figure 3.1 illustrates a sequence diagram that shows the process of creating a new box in MHyPro. In order to create a new box the MEX function `MHyPro` has to be called with the parameters *'box'* and *'newBox'*. The MEX function calls then the `process` function of the class `Box`, which can call the `newBox` function. Finally, the `newBox` function calls the HyPro constructor for boxes. The constructor returns a pointer to the new box. The box constructor in MHyPro calls then the class handler in order to cast the pointer so Matlab can use it. The class handler returns a new pointer which is than forwarded back to Matlab. In Matlab the new box object stores the pointer as its parameter so it can identify itself when calling another functions. The over head caused by the wrapper will be measured in the next chapter.

In the next chapter the wrapper will be used for the comparison of the performance and the precision with Cora.

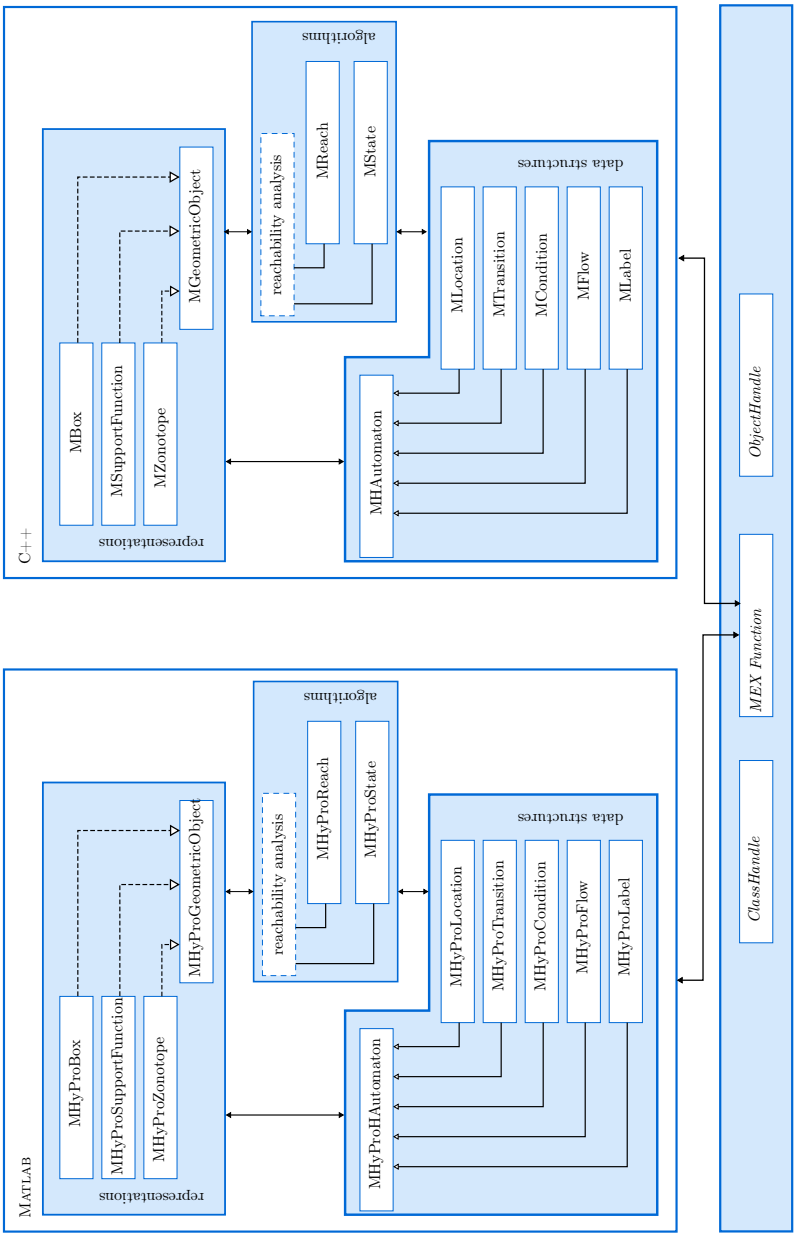


Figure 3.2: Structure of MHYPRO.

Chapter 4

Experimental Results

In this chapter the efficiency and precision of MHyPro and Cora is compared. In the first part of this chapter, the flowpipe construction of both tools is compared. In the second part, a set of safety specifications is defined and the performance of the reachability analysis algorithms is compared. Subsequently, the performance of operations on two state set representations is compared. Finally in the last part, the overhead caused by Matlab for MHyPro is determined.

4.1 Comparing Flowpipe Construction Algorithms

In order to compare the performance of the Matlab-wrapper for HyPro and Cora, nine different benchmarks were selected. The nine benchmarks are commonly known. The most basic benchmark is the *bouncing ball* (bb) benchmark modelling a ball that was drooped from a predefined height that bounces off the ground several times.

A real world benchmark is the *filtered oscillator* (fo$X$) benchmark that models a two-dimensional switched oscillator and a fourth order filter that smooths a signal. Here, three instances of the benchmark are considered that differ in the number of variables. The X behind the name indicates how many variables the oscillator considers. Since the running time depends on the number of dimensions, this benchmark is suitable to determine how well the tools scale.

An another real world benchmark is the *rod reactor* (rr) benchmark that models a reactor tank that contains a liquid, and two rods that can be used to cool down the temperature of the liquid. The rods have different cooling dynamics and only one of the rods can be used for cooling simultaneously. The rods can be exchanged, as soon as a defined amount of time has passed.

The *spacecraft rendezvous* (sr) benchmark was taken from [CM17] and models the motion of a spacecraft in orbit with the dynamics derived from Kepler's law. The fact that this benchmark was used in the Applied Verification for Continuous and Hybrid Systems (ARCH) competition in 2018, makes the benchmark relevant.

The *switching system* (sw) is an artificial benchmark consisting of five locations where each location is labelled with different dynamics. Although it is a purely academic benchmark, it is still relevant as it requires precise computations and is therefore challenging.

The *two tanks* (tt) benchmark is a classic benchmark. It models two connected tanks whose inflows are controlled by valves. As soon as the fill level in one of the

tanks is too low, the tank is refilled.

The *vehicle platoon* (vp) is an another real world benchmark that models a platoon of three vehicles guided by a leader vehicle.

The hybrid automata modelling the particular benchmarks can be found in the appendix (A.1). The abbreviations written in braces behind the names of the benchmarks will be used in all tables from now on. All computations were performed on an Intel Core i5 (2.6 GHz) processor with 8 GB RAM. The timeout was set either to 105 s or to 195 s depending on the difficulty of the specifications, however, MATLAB needs to be started every time which takes about 15 seconds thus for each benchmark the tools had about 90 s (180 s) to solve the particular benchmark.

4.1.1 Comparing Flowpipe Computations with Initial Settings

First, the reachability analysis algorithms of MHyPro and Cora will be compared without safety specifications. The benchmarks were run with initial settings that are listed in Table 4.1 and 4.2.

Benchmark	Set Repr.	Time Step Size [s]	Time Horizon [s]	Jumps
bb	box	1×10^{-2}	4	2
fo4	box	5×10^{-2}	4	5
fo8	box	5×10^{-2}	4	5
fo16	box	5×10^{-2}	4	5
sr	box	1×10^{-2}	20	10
sw	sf	1×10^{-3}	1	5
rr	box	1×10^{-2}	15	5
tt	box	1×10^{-2}	2	2
vp	sf	2×10^{-2}	12	20

Table 4.1: Initial strategies used for the reachability analysis with MHyPro. The abbreviation *sf* signifies support functions. All strategies use aggregation and non-uniform bloating.

Benchmark	Time Step Size [s]	Time Horizon [s]	Zonotope Order	Taylor Terms
bb	1×10^{-2}	4	1	1
fo4	5×10^{-2}	4	2	2
fo8	5×10^{-2}	4	2	2
fo16	5×10^{-2}	4	200	100
sr	1×10^{-2}	20	1	1
sw	1×10^{-3}	1	4	1
rr	1×10^{-2}	15	1	1
tt	1×10^{-2}	2	2	1
vp	2×10^{-2}	12	5	1

Table 4.2: Initial strategies used for the reachability analysis with Cora.

In order to make the results for both tools as comparable as possible, the initial settings for both tools have to be comparable. Therefore, the time-step size, the initial set and the time horizon for each of the benchmarks is equal for both tools. Moreover, one has to pay attention to two important facts. First, one has to attend that the time horizon in MHyPro is local and not global like in Cora. To add global time horizon in MHyPro, each model has an additional variable (clock) that measures the global time. The local time is equal to the global time in order to guarantee that MHyPro computes the same number of reachable states as Cora. Since an additional variable increases the dimension of every benchmark in MHyPro by one, the running times are slightly slower than when running the benchmarks without the global clock. Second, Cora does not consider the jump depth. Therefore, in order to guarantee that the number of flowpipes computed by MHyPro and Cora is equal, the jump depth for the MHyPro models was adjusted appropriately.

Besides the time-step size and the time horizon, for MHyPro one has to determine the state set representation and the jump depth. The choice of the state set representation depends on the precision of the reachability analysis that is required by a benchmark. In context of this comparison, only boxes and support functions were considered for MHyPro as state set representation. Since support functions are computationally more expensive than boxes, they are only chosen if the benchmark requires precise computations. The switching system and the vehicle platoon benchmarks are examples of such benchmarks. All remaining benchmarks use boxes as the initial state set representation.

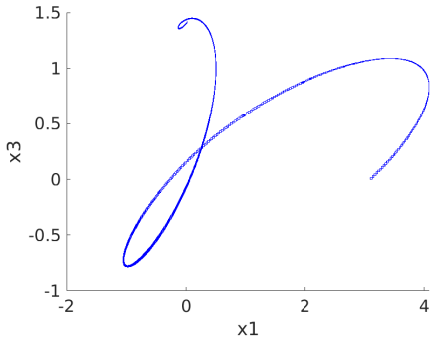
For Cora one has to determine the zonotope order and the number of considered Taylor terms. Usually, for each benchmark there exists a minimal value for the parameters. Further decrease of one of the parameters make the result of the reachability analysis not reasonable. For example if the zonotope order for the switching system is less than four, then the computed flowpipe is very imprecise as Figure 4.1 shows. However, the determination of the two parameters is not always that easy as the filtered oscillator 16 benchmark has shown. Many values have been tested but no of the tested values achieved reasonable results. Therefore, the parameter values were set to values that, by experience, are good default values.

Benchmark	Run Time [s] MHyPro	Run Time [s] Cora
bb	0.0214	0.6915
fo4	0.0063	2.0818
fo8	0.0046	10.7650
fo16	0.0071	memout
sr	0.0577	1.1910
sw	25.6838	1.2875
rr	0.1265	15.4047
tt	0.0091	0.6280
vp	9.1345	4.8854

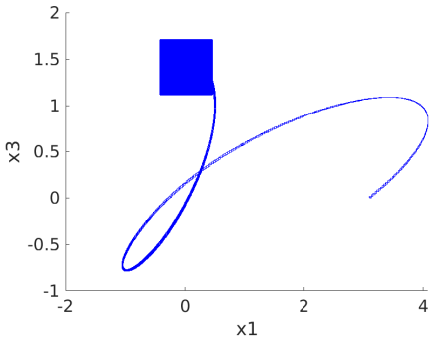
Table 4.3: Results achieved by MHyPro and Cora using the initial settings.

The timeout for this computations was set to 105 s. The results obtained by the two tools are shown in Table 4.3.

MHyPro could compute all reachable sets within the 105 s, while Cora was not



(a) Flowpipe computed by CORA with initial settings.



(b) Flowpipe computed by CORA with modified initial settings. Here, the zonotope order was set to three instead of four.

Figure 4.1: Flowpipes computed by CORA with different zonotope orders for the switching system benchmark.

able to compute the flowpipes for the filtered oscillator 16 benchmark. Except for the switching system and the vehicle platoon benchmarks, MHYPRO is clearly faster than CORA. One reason for MHYPRO's fast computations is the fact that MHYPRO directly calls HYPRO which is implemented in C++ that is faster than MATLAB. Now each of the benchmarks is discussed in more detail.

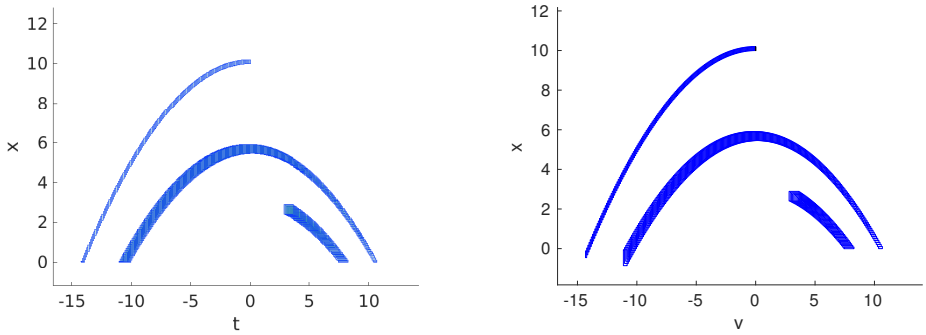
First the bouncing ball benchmark is considered. MHYPRO was about thirty times faster than CORA, although the computed flowpipes are nearly exactly the same as Figure 4.2 illustrates. The left figure illustrates the flowpipe computed by MHYPRO and the right one the flowpipe computed by CORA. When one overlaps the flowpipes, then nearly only one flowpipe is visible.

As already mentioned, the benchmark models a ball that was dropped from a predefined height. When the distance to the ground (x) is exactly zero, the ball bounces. An invariant that guarantees that x is always greater or equal zero, and a guard, that enables bouncing (i.e. taking a discrete transition) only if x is equal zero, were introduced to guarantee this behaviour. When the flowpipes depicted in Figure 4.2b are considered more closely, one can see that the flowpipes computed by CORA violate the invariant. This phenomenon is discussed now in detail.

In order to compute a single flowpipe, CORA iteratively computes single segments as long as the current reachable set is within the invariant, and the time horizon was not exceeded. Hence, in each iteration CORA have to check if the invariant is still satisfied. For this purpose CORA first projects the current segment on the dimensions that the invariant concern. The result of this projection is a set of intervals. A segment is within the invariant, as long as all minima of the obtained intervals are smaller than the upper bound of the invariant. This guarantees that all flows are captured. However, this causes the computation of a segment that lies completely outside the invariant.

The dynamics of this benchmark is defined as follows

$$\begin{aligned} x'(t) &= v(t) \\ v'(t) &= -9.81 \end{aligned} \tag{4.1}$$



(a) Flowpipes computed by MHyPro for the bouncing ball benchmark with initial settings.

(b) Flowpipes computed by Cora for the bouncing ball benchmark with initial settings.

Figure 4.2: Reachability analysis of the bouncing ball benchmark computed by Cora (blue) and MHyPro (green).

In order to find when the ball should bounce, the following equation has to be solved

$$0 = \underbrace{\frac{1}{2} \cdot -9.81 \cdot t^2 + v_0 \cdot t + x_0}_{=x(t)} \tag{4.2}$$

where v_0 and x_0 are the initial values for the variables v and x. The initial value of x is the interval $[10, 10.2]$, hence, the solution for Equation 4.2 is the time interval $t = [1.427, 1.442]$. This means that within this time interval the ball should bounce for the first time. Now the segments lying near this time interval are considered. Figure 4.3 shows the corresponding segments. Note that the x-axis is now t and not v as in the previous figures. Therefore, one can also see the first segments of the second flowpipe. The red dots depict the infima of the segments and are labeled with their coordinates. The red line is the simulation of the benchmark starting with the initial set $t = 0$ and $x = 10.1$. The black point is the point where the ball bounces off the ground (this coordinates are marked with yellow background). The last segment that completely lies within the invariant is the one starting at $t = 1.41$. Although the suprema of the next two segments are already negative, and the segment at $t = [1.43, 1.44]$ already contains the point where the ball bounces, Cora has to compute the next segment since the infima of the segment is still positive. Otherwise, if the algorithm would already stop the computation, the algorithm would disregard the reachable area in the time interval $t = [1.43, 1.44]$, that still fulfills the invariant. This would make the algorithm faulty. The infimum of the next segment is still greater than zero (0.03), hence the next segment is computed. The infimum of the nest segment if finally negative (-0.11) and Cora stops computing further segments, however, although the last segment lies completely outside the invariant, Cora keeps it in the set of reachable sets of states.

The segments that lie partially outside the invariant and intersect a guard, can be removed when the parameter `intersectInvariant` is set. The segments that partially lie outside the invariant are then over-approximated by polytopes and subsequently intersected with the invariant. For the bouncing ball benchmark only the

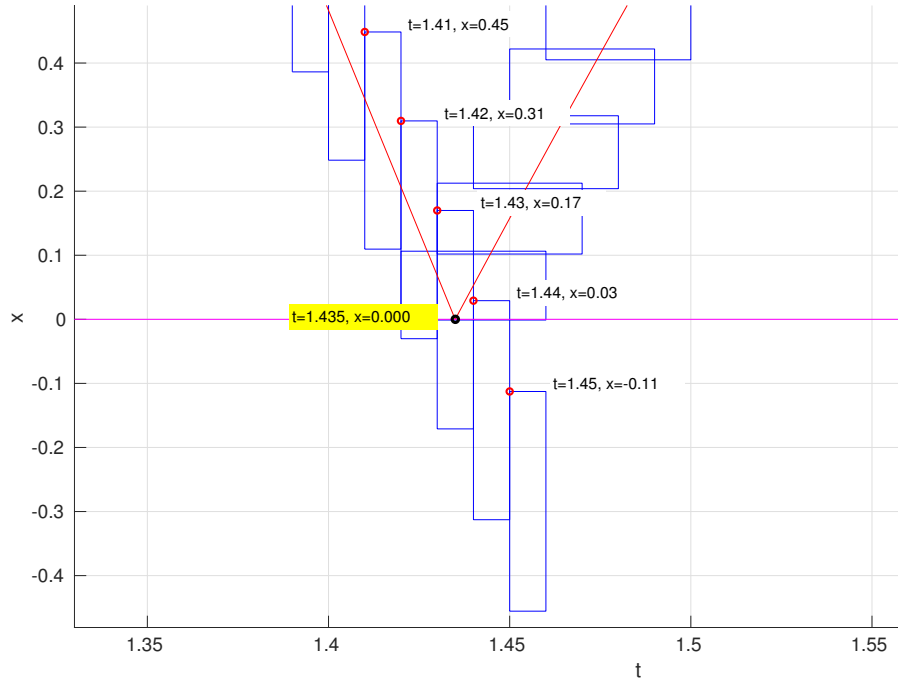


Figure 4.3: Last segments of the first flowpipe of the bouncing ball benchmark computed by CORA with the initial settings. The magenta colored line depicts the invariant that guarantees that x is greater than zero. One can clearly see that CORA computes one segment that lies completely outside the invariant.

segments lying in the time interval $t = [1.42, 1.44]$ intersect the guard ($x = 0$). The last segment is not considered. Therefore, the resulting flowpipes look as depicted in Figure 4.4. The segment that does not intersect the guard was not removed. Note that if the segments that partially intersect the invariant do not intersect any guard set, the procedure does not change any of the segments.

Since CORA provides various possibilities to represent invariants, for example by intervals, zonotopes, halfspaces, and Taylor models, for all of the representations it was tested if they can solve the problem. Unfortunately, the results obtained using the different representations of invariants, are the same as for invariants defined by polytopes.

Next the filtered oscillator benchmarks are considered. According to Table 4.3, MHYPRO again could compute the flowpipes faster than CORA, and could solve the filtered oscillator 16 benchmark while CORA was only able to compute the set of states reachable within the first 1.05 seconds. The reason why CORA cannot solve the filtered oscillator 16 benchmark is the involving computation of the intersection of the reachable sets with the set of guards. In order to accelerate the process CORA first checks if there are any potential segments intersecting a guard set. If so, then the segments are over-approximated by polytopes and intersected with the guards.

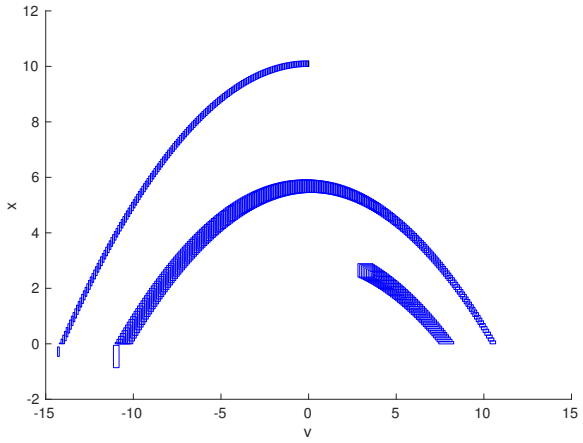
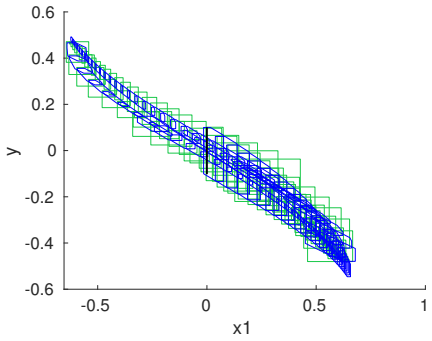


Figure 4.4: Flowpipe of the bouncing ball benchmark computed by CORA with initial setting and the set parameter `invariantIntersect`.

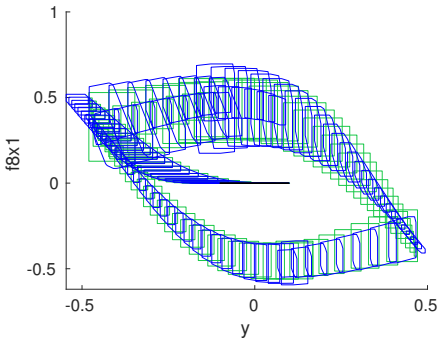
For this benchmark CORA needs to calculate the guard intersection seven times. For the filtered oscillator 4, on average, CORA needs about 6.5 ms to estimate which segments intersect the guard and about 0.2441 s to calculate the intersection. For the filtered oscillator 8, on average the estimation of the intersecting segments takes about 8.6367 ms and the calculation of the intersection takes about 0.7575 s, which is about three times longer than for the filtered oscillator 4 benchmark. For the filtered oscillator 16, the computation of the first intersection takes already about 10 s and the computation of the second intersection causes an out-of-memory error after about 18 minutes. Therefore, it is difficult for CORA to verify this benchmark. The flowpipe computed by MHYPRO for the filtered oscillator 16 benchmark can be found in the appendix (Figure A.8a).

The flowpipes computed for the filtered oscillator 4 benchmark are considered now in detail. The flowpipes are illustrated in Figure 4.5a. The green flowpipes were computed by MHYPRO, the blue flowpipes were computed by CORA. The green flowpipes are less precise than those computed by CORA. One can clearly see the particular segments represented by boxes. The flowpipes computed by CORA are much smoother, however, this comes with the price of higher running time that is about 330 times higher than for MHYPRO. A bit different result was obtained for the filtered oscillator 8 benchmark (see Figure 4.5b). The flowpipes computed by MHYPRO are still not that smooth as those computed by CORA, but at some points the flowpipes computed by MHYPRO are tighter than CORA's. Moreover, for both benchmarks the flowpipes computed by CORA again violate the invariants. Figure 4.6 illustrates the flowpipes computed by MHYPRO (left) and CORA (right) starting from the initial location without taking any discrete transitions while the time horizon was set 2 s. The magenta line marks the invariant. All states that are above this line satisfy the invariant. The flowpipe computed by CORA clearly violates the invariant. The flowpipe computed by MHYPRO also violates the invariant, however, this is caused by the over-approximation of the boxes, whereas CORA computes a segment that lies completely outside the invariant.

Spacecraft rendezvous is the next considered benchmark. The flowpipes computed

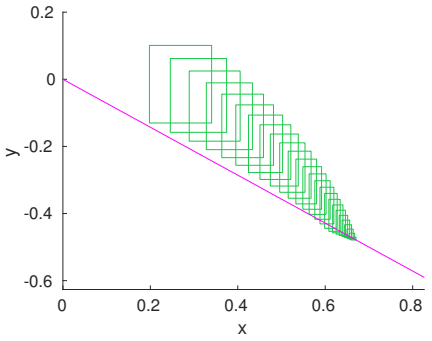


(a) Reachability analysis of the filtered oscillator 4 benchmark performed by CORA (blue) and MHYPRO (green) with the initial settings.



(b) Reachability analysis of the filtered oscillator 4 benchmark performed by CORA (blue) and MHYPRO (green) with the initial settings.

Figure 4.5: Reachability analysis results for the filtered oscillator 4 and 8 computed with initial settings.



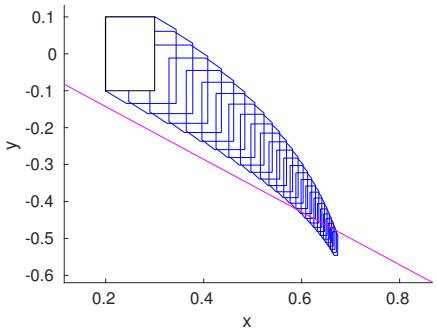
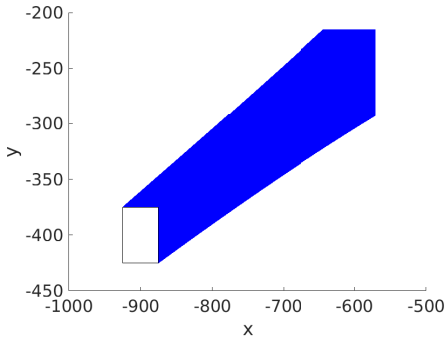
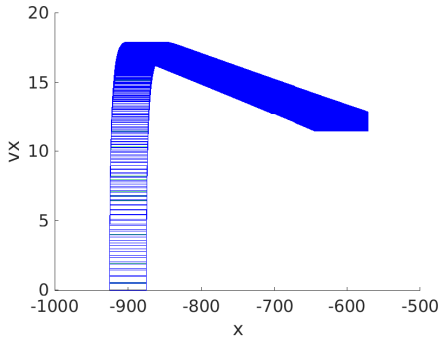


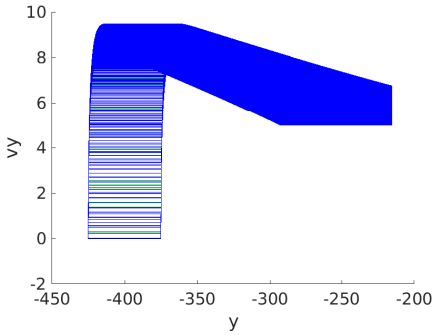
Figure 4.6: Reachability analysis of the filtered oscillator 8 benchmark computed by MHYPRO (left) and CORA (right) with initial settings and time horizon of 2 s. The green line depicts the guard of the initial location. All states that are above this line satisfy the invariant.



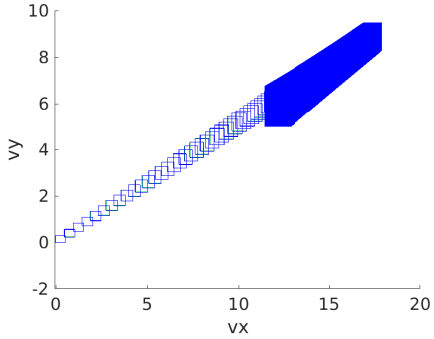
(a) Projection on the $x-y$-dimensions.



(b) Projection on the $x-vx$-dimensions.



(c) Projection on the $y-vy$-dimensions.

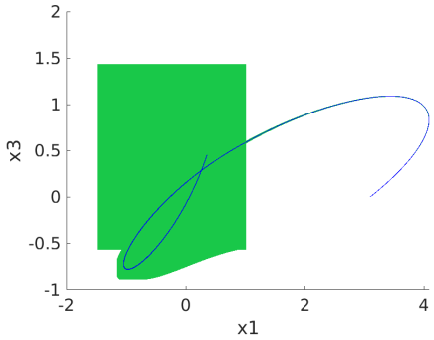


(d) Projection on the $vx-vy$-dimensions.

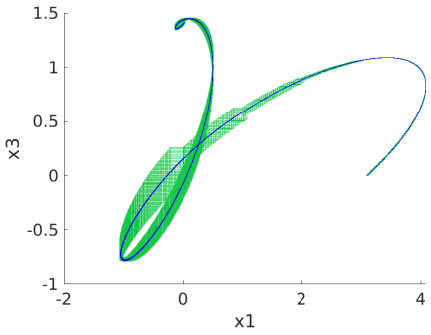
Figure 4.7: Reachability analysis of the spacecraft rendezvous benchmark with the initial settings. The four figures show the flowpipes projected on different dimensions. Since first the flowpipes computed by MHyPro were plotted (green) and then those computed by Cora, the flowpipes are mostly blue. However, the plotting order does not matter for this benchmark. For both plotting orders the flowpipes are nearly equal.

by MHyPro and Cora are nearly equal. Figure 4.7 illustrates the overlapped flowpipes computed by MHyPro and Cora. Due to the fact that first the flowpipes computed by MHyPro were plotted (green) and then those computed by Cora (blue), the flowpipes are mostly blue. However, the plotting order does not matter for this benchmark (only the color changes). The upper left Figure shows the projection of the flowpipes on the $x-y$-dimensions, the upper right on the $x-vx$-dimensions, the lower left on the $y-vy$-dimensions, and finally the lower right on the $vx-vy$-dimensions. Since nearly no green color is visible in any of the projections, the flowpipes are nearly equal in every dimension.

The switching system benchmark is considered next. For MHyPro it is one of two benchmarks using support functions as state set representation. The reason for this choice is the fact that the over-approximation caused by boxes is enormous as Figure 4.8 shows. In both figures one can see the flowpipe computed by Cora with the initial settings (blue), and the one computed by MHyPro in green. The settings for MHyPro used to compute the flowpipe illustrated in the left figure, are the



(a) Flowpipe computed by MHyPro (green) and Cora (blue). MHyPro was run with modified initial settings. Instead of support functions boxes were used as state set representation. The time horizon was set to 0.5 s.



(b) Flowpipe computed by MHyPro (green) and Cora (blue) with initial setting.

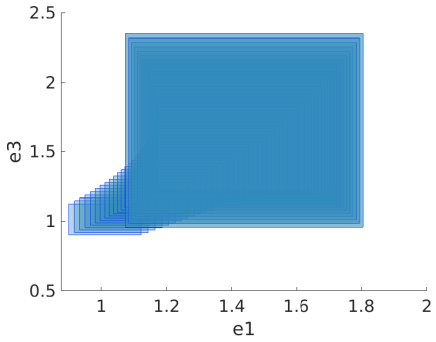
Figure 4.8: Reachability analysis results for the switching system benchmark.

same as the initial settings except for the fact that boxes were used as the state set representation. Moreover, the time horizon was set to 0.5 s instead of 1 s because a larger time horizon causes out-of-memory errors. The flowpipes depicted in the right figure were both computed using the original initial settings. The flowpipes computed by MHyPro are still coarse in comparison to those computed by Cora, however, the result is better than the obtained for boxes, since here the time horizon was set to 1 s.

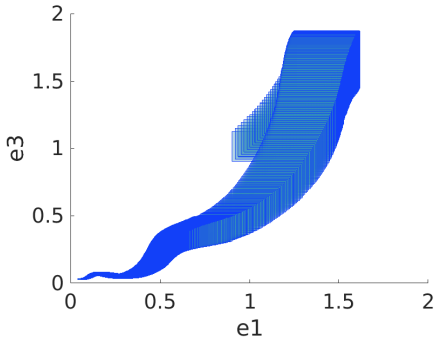
Similar problem can be observed for the vehicle platoon benchmark and therefore also for this benchmark the initial settings use support functions as state set representation. Figure 4.9a shows the flowpipe computed by MHyPro with the same settings as the initial but with boxes as state set representation and the time horizon reduced to 1 s (originally it is set to 12 s). The flowpipes depicted on the right (Figure 4.9b), illustrates the flowpipes computed by MHyPro with the original initial settings. However, for this benchmarks also Cora has problems computing precise flowpipes. Figure 4.9c depicts the flowpipes computed by MHyPro (green) and Cora (blue). First the flowpipes computed by Cora were plotted and then those computed by MHyPro.

Now the rod reactor benchmark is shortly considered. The flowpipes computed by Cora clearly violate the invariant $x \leq 550$, however, the remaining flowpipes are exactly the same for both tools as Figure 4.10a shows.

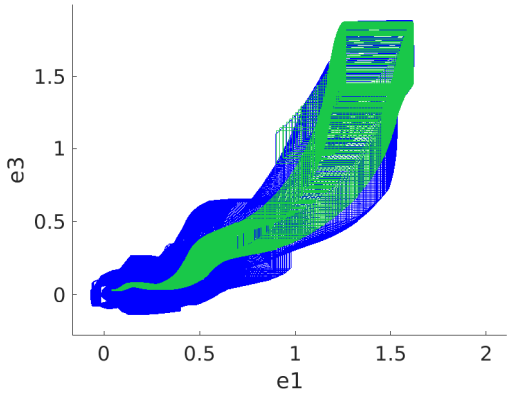
Finally the two tanks benchmark is considered. The results achieved by both tools are very different as Figure 4.10b shows. The flowpipes computed by MHyPro are very wide in comparison to the flowpipes computed by Cora. Similarly as for other benchmarks, Cora again violates the invariant ($x \geq -1$), while MHyPro precisely stops its computations as soon as x is equal -1.



(a) Flowpipe computed by MHyPro with modified initial settings. Instead of support functions here boxes were used as state set representation. The time horizon was set here to 1 s.

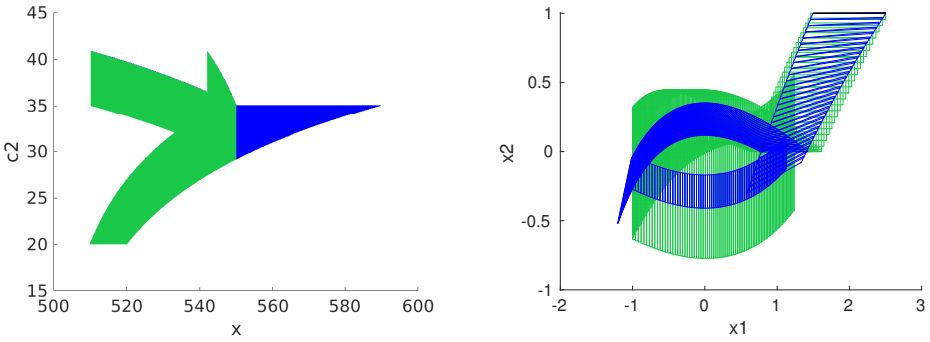


(b) Flowpipe computed by MHyPro with the original initial settings.



(c) Flowpipes of the vehicle platoon benchmark computed by Cora (blue) and MHyPro (green).

Figure 4.9: Reachability analysis results for the vehicle platoon benchmark.



(a) Reachability analysis of the rod reactor benchmark computed by CORA (blue) and MHYPRO (green) computed with the initial settings.

(b) Reachability analysis of the two tanks benchmark computed by CORA (blue) and MHYPRO (green) computed with the initial settings.

Figure 4.10: Reachability analysis results for the rod reactor (left) and two tanks (right) benchmarks performed by CORA (blue) and MHYPRO (green).

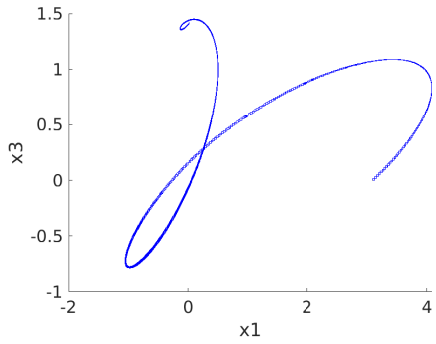
4.1.2 Comparing Safety Verification

The results obtained for the initial strategies have indicated that both tools have strength and weaknesses. For the most benchmarks, MHYPRO was faster than CORA, but most of the computed flowpipes were less precises than those computed by CORA. However, one has to recall that the settings used for the computations were initial. The choice of the parameters has great impact on the results of the reachability analysis. Therefore, before the results for the safety verification will be presented, first the influence of the particular parameters will be explained.

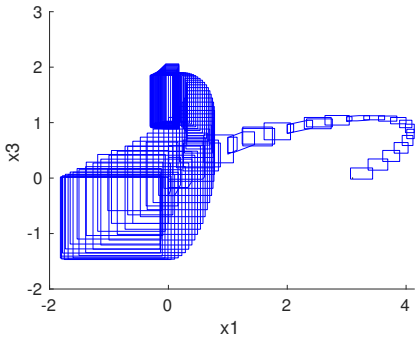
4.1.3 Influence of the Parameters on the Reachability Analysis

A parameter that has a lot of impact on the reachability analysis result is the time-step size. A large time-step size causes great over-approximations but the time needed for the computations decreases. The impact of this parameter is illustrated on the example of the switching system benchmark. Figure 4.11 shows flowpipes computed by CORA using different time-step sizes while the remaining parameters are as defined by the initial settings. For the computation of the left flowpipe the initial settings were used with the time-step size of 0.001 s. For the computation CORA needed 1.2875 s. For the computation of the right flowpipe the time-step size was increased to 0.01 s. For this computation, CORA needed 0.561 61 s, i.e., about 2.3 times less time than with the initial settings. However, one can clearly see that the flowpipe is less precise than the left one. The flowpipe is thicker and the particular segments are clearly visible.

CORA requires the determination of the maximal zonotope order. The higher the zonotope order, the higher number of generators was used for the definition of the zonotope. Often operations like the Minkowski sum, can increase the order of a zonotope. Unfortunately, the higher the zonotope order, the more time is needed to perform operations on the zonotope. Therefore, if the order than the maximal zonotope order, the corresponding zonotope is over-approximated, as tight as possible, by zonotopes of smaller order [KSA17]. This over-approximation decreases the running



(a) Flowpipe computed by CORA for the switching system benchmark using the initial strategy with time-step size set to 0.001 s.



(b) Flowpipe computed by CORA for the switching system benchmark using the initial strategy with modified time-step size. Here, the time-step size was set to 0.01 s

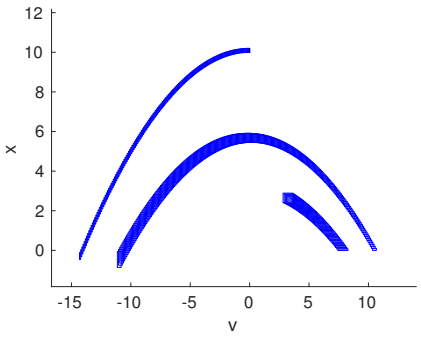
.

Figure 4.11: Impact of the time-step size on the reachability analysis results shown on the example of the switching system benchmark.

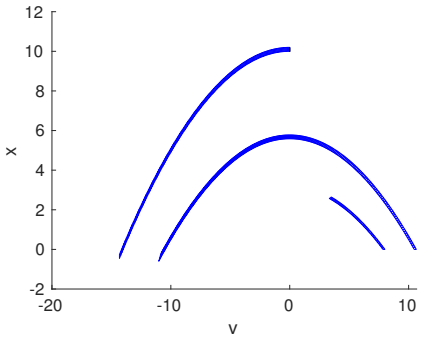
time, but also decreases the precision of the resulting flowpipe. The impact of this parameter can be shown on the example of the bouncing ball benchmark (see Figure 4.12). With the initial settings (zonotope order is equal one) CORA needs about 0.679 23 s to compute the flowpipes. When the zonotope order is set to 20, CORA needs 1.4655 s, i.e., about two times longer than with the initial settings. However, the resulting flowpipes are clearly narrower.

Another parameter required by CORA is the one concerning the number of considered Taylor terms. Consider the vehicle platoon benchmark illustrated in Figure 4.13. The blue flowpipe was computed with the initial settings, the green one with modified number of Taylor terms. Originally for initial settings, the parameter is set to 1, here, the parameter was set to 100. The resulting flowpipe is narrower than the blue one. For the computation of the flowpipe with the modified initial settings CORA needed about 1.3 times longer than with the initial settings (4.8854 s with the modified settings and 6.3718 s with the original settings).

For MHYPRO, a great impact on the results of the reachability result has the clustering parameter. All initial strategies use aggregation, since clustering increases the running time exponentially. However, when clustering is used, the computed flowpipes are tighter. Figure 4.14 shows the effect of clustering on the example of the switching system benchmark. The flowpipe on the left side was computed with the initial setting, the one on the right side with modified initial settings were clustering was set to three. One can see that the flowpipe on the right side is smoother than the one on the left. For the computations with the initial settings, MHYPRO needed 25.6838 s while for the computations with the modified settings 26.7813 s were needed.



(a) Flowpipe computed by CORA for the bouncing ball benchmark using the initial settings with zonotope order set to one.



(b) Flowpipe computed by CORA for the bouncing ball benchmark using the initial settings with modified zonotope order. Here, the zonotope order was set to 20

.

Figure 4.12: Impact of the zonotope order for the bouncing ball benchmark.

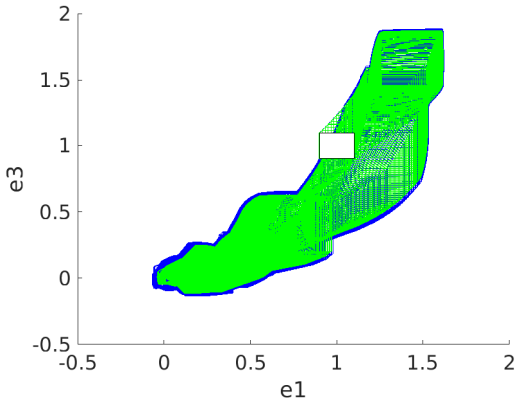
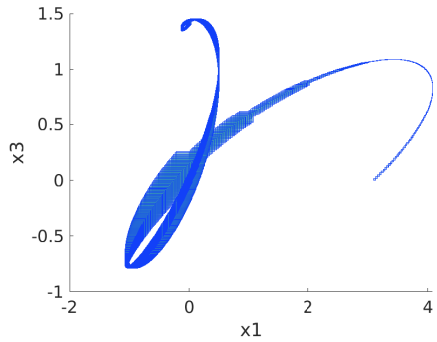
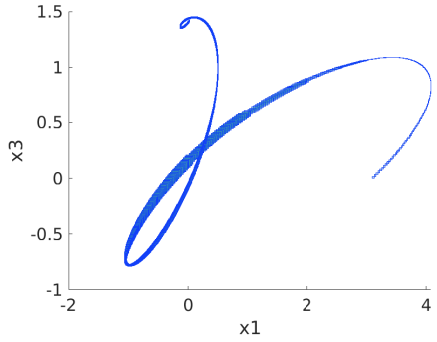


Figure 4.13: Impact of the number of considered Taylor terms tested on the vehicle platoon benchmark. The blue flowpipe was computed with the initial settings. The green one with modified number of considered Taylor terms. Here the parameter was set to 100.



(a) Flowpipe computed by MHyPro for the switching system benchmark using the initial settings with aggregation.



(b) Flowpipe computed by MHyPro for the switching system benchmark using the initial settings with clustering (maximal three segments)

Figure 4.14: Impact of clustering on the reachability analysis shown on the switching system benchmark.

4.1.4 Verifying Easy Safety Specifications

The parameters have great impact on the results of the reachability analysis as the examples presented in the previous section have shown. In this and the next subsections, two sets of safety specifications will be considered that should be verified by the both tools. The sets differ in the difficulty of the specifications. The specifications are constraints of the form $\sum_i a_i \cdot x_i \leq c_i$ where $a_i, c_i \in \mathbb{R}$ and x_i are the state variables. The first type of specifications (easy specifications) are those that were given in the model files of the benchmarks. For the hard specification, the right hand side of the constraints were changed such that Cora nearly could not verify the specification when the zonotope order was set to 200 and the number of Taylor terms was set to 100 for every benchmark (the time-step size was defined as for the initial settings). The obtained specifications are listed in Table 4.4.

Since Cora does not provide a verification algorithm, it was implemented by us in Matlab. The verification algorithm for MHyPro was also implemented in Matlab and has the same concept as the one for Cora. Thus, the performance of both algorithms is comparable.

For the verification of the specifications, 42 different strategies for MHyPro and 36 different strategies for Cora were run to test the different settings. The time horizon and the initial sets are for both tools the same. The jump depth for MHyPro is the same as for the initial settings. The remaining parameters vary. All strategies for both tools are listed in Tables A.1 and A.2 that can be found in the appendix.

The Tables 4.7 and 4.8 contain the results achieved by MHyPro and Cora with the fastest strategies. The corresponding settings can be found in Tables 4.5 and 4.6. Overall one can see that the strategies used for the verification of the easy specifications required more time than for the initial strategies. There are two reasons for the increased running times. First, the verification algorithm had to be run in

Benchmark	Easy Specification	Hard Specification
bb	$v \leq 10.700$	$v \leq 10.614$
fo4	$y \leq 0.500$	$y \leq 0.464$
fo8	$y \leq 0.500$	$y \leq 0.469$
fo16	$y \leq 0.500$	$y \leq 0.464$
sr	$vx \leq 18 \wedge vy \leq 10$	$vx \leq 17.868 \wedge vy \leq 9.442$
sw	$x_3 \leq 1.5$	$x_3 \leq 1.488$
rr	$c_2 \leq 41.1$	$c_2 \leq 40.860$
tt	$x_2 \geq -0.7$	$x_2 \geq -0.507$
vp	$e_1 \leq 1.7$	$e_1 \leq 1.636$

Table 4.4: Easy and hard safety specifications used for the efficiency tests.

order to verify the specifications. Second, most of the specifications required tighter flowpipes in order to satisfy the specifications, therefore, most of the strategies used here use smaller time-step size which increases the running time.

Benchmark	Set Repr.	Time-Step Size [s]	Clust.
bb	box	5×10^{-3}	0
fo4	box	2×10^{-2}	0
fo8	box	2×10^{-2}	0
fo16	box	2×10^{-2}	0
sr	box	2×10^{-2}	0
sw	sf	5×10^{-3}	3
rr	box	1×10^{-1}	3

Table 4.5: Fastest strategies for the verification of the easy specification for MHyPro. Since no of the strategies could verify the two tanks and the vehicle platoon benchmarks, they are not listed in the table.

The results obtained by MHyPro show that on average the verification algorithm needed 81.6132 % of the running time. This high percentage can be explained by the fact that the algorithm is implemented in Matlab. MHyPro could not verify the vehicle platoon and the two tanks benchmarks within 105 s. Even an increase of the timeout to five minutes was not enough for MHyPro to solve these benchmarks. However, except for the switching system benchmark, MHyPro was clearly faster than Cora, on average about 12 times.

Cora could not verify the filtered oscillator 16 benchmark, but this is not that surprising since already the search for initial settings for this benchmark was not successful. Note that the percentage of the time spent on the verification is far lower than that of MHyPro. On average the verification algorithm needed 12.8281 % of the running time. This acknowledges, that the overhead caused by Matlab cannot be neglected.

Now the flowpipes of the benchmarks will be shortly discussed. Since easy specifications do not require that precise flowpipes and here only the fastest strategies are

Benchmark	Time-Step Size [s]	Taylor Terms	Zonotope Order
bb	2×10^{-1}	100	200
fo4	5×10^{-2}	2	2
fo8	1×10^{-2}	100	1
sr	2×10^{-1}	100	200
sw	1×10^{-2}	10	20
rr	5×10^{-1}	10	20
tt	5×10^{-2}	100	200
vp	1×10^{-1}	10	20

Table 4.6: Fastest strategies for easy specification for CORA. Since no of the strategies could verify the benchmark filtered oscillator 16, it is not listed in the table.

Benchmark	Reach. Time [s]	Verif. Time [s]	Running Time [s]
bb	0.0294	0.0985	0.1280
fo4	0.0194	0.1294	0.1488
fo8	0.0331	0.3366	0.3697
fo16	0.0230	64.8776	64.9006
sr	0.0310	0.1503	0.1812
sw	3.2736	9.5118	12.7854
rr	0.1202	0.1731	0.2932

Table 4.7: Running times achieved by MHYPRO for easy specifications with the fastest strategies. The second column contains the times needed for the reachability analysis. The third column contains the times needed for the verification of the specifications. The last column contains the total times.

considered, the results for some benchmarks are more imprecise in comparison to those achieved with the initial settings. A good example for this circumstance is the bouncing ball benchmark. Figure 4.15 illustrates the overlapped flowpipes computed by CORA (blue) and MHYPRO (green). The red rectangle symbolizes the bad states. Neither of the flowpipes intersects with the red rectangle, however, the flowpipe computed by CORA is coarser than the one computed with the initial settings. The reason for this result is the fact that the time-step size defined by the used strategy was twice as big as defined by the initial settings (2×10^{-2} s instead of 1×10^{-2} s). Therefore, the segments of the flowpipes are bigger causing greater over-approximations than for the initial settings. Even though the strategy MHYPRO used time-step size of S5 $\times 10^{-3}$ s, MHYPRO was ten times faster than CORA.

Similar results were achieved for the rod reactor and the spacecraft rendezvous benchmarks. The flowpipes are more imprecise than those computed with the initial settings, as the time-step size for both strategies was increased. The corresponding flowpipes can be found in the appendix (Figure A.9).

The results for the reachability analysis of the filtered oscillator 4 benchmark are illustrated in Figure 4.16a. The figure shows the flowpipes computed by CORA (blue) and MHYPRO (green). The flowpipes computed by CORA for the filtered oscillator

Benchmark	Reach. Time [s]	Verif. Time [s]	Running Time [s]
bb	1.0093	0.2008	1.2101
fo4	3.0175	0.1767	3.1942
fo8	8.4786	0.8722	9.3508
fo16	-	-	-
sr	0.4397	0.1757	0.6154
sw	1.2161	0.1308	1.3469
rr	1.2198	0.0686	1.2884
tt	1.0754	0.0322	1.1076
vp	4.8126	0.2496	5.0622

Table 4.8: Running times achieved by CORA for easy specifications with the fastest strategies. The second column contains the times needed for the reachability analysis. The third column contains the times needed for the verification of the specifications. The last column contains the total times.

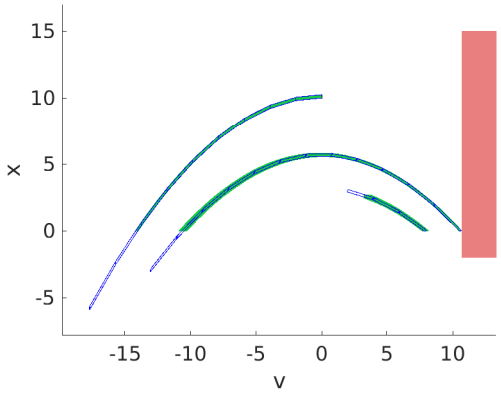


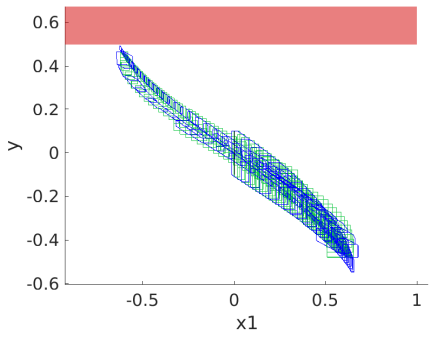
Figure 4.15: Reachability analysis of the bouncing ball benchmark computed by CORA (blue) and MHYPRO (green) for the easy specification with the fastest strategy.

4 benchmark are exactly the same as those computed with initial settings, since it turned out that the initial strategy was the feasts one for this benchmark. The flowpipe computed by MHYPRO for this benchmark is only slightly tighter than the one computed with initial settings, since only the time-step size was slightly changed (initially 5×10^{-1} s, now 2×10^{-2} s).

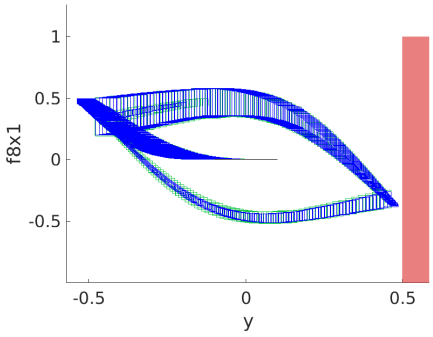
The improvement for the filtered oscillator 8 benchmark is more visible as Figure 4.16b illustrates. Both flowpipes are clearly tighter than those computed with the initial settings. Again, the reduced time-step size is responsible for this improvement.

Since among the 36 strategies in the set for CORA, there was no strategy that could verify the filtered oscillator 16 benchmark within the 105 s, only the flowpipe computed by MHYPRO was plotted and can be found in the appendix (Figure A.8b).

An improvement was also achieved for the switching system benchmark by MHYPRO. The flowpipes computed by MHYPRO (green) and CORA (blue) are illustrated in Figure 4.17. The flowpipes computed by MHYPRO are smoother than those computed



(a) Reachability analysis of the filtered oscillator 4 performed by CORA (blue) and MHYPRO (green) with strategy for the easy specifications.



(b) Reachability analysis of the filtered oscillator 8 performed by CORA (blue) and MHYPRO (green) with strategy for the easy specifications.

Figure 4.16: Reachability analysis results computed with the strategies for the easy specifications for the filtered oscillator benchmarks.

with the initial settings. The strategy used by MHYPRO for this computations used greater time-step size (initially 1×10^{-3} s, here 5×10^{-3} s), but instead of aggregation, clustering was used. The maximal number of segments for the clustering was set to three.

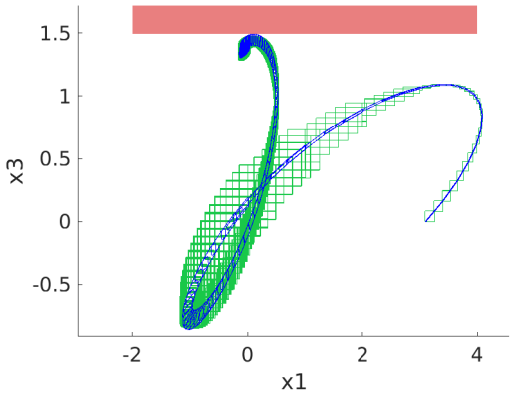


Figure 4.17: Reachability analysis of the linear switching systems benchmark computed by CORA (blue) and MHYPRO (green) for easy specification.

Due to the fact that MHYPRO could not verify the two tanks and the vehicle platoon benchmarks only the flowpipes computed by CORA were plotted which can be found in the appendix (Figure A.10).

4.1.5 Verifying Hard Safety Specifications

Since hard specifications require more precise computations, the timeout was increased to 195 s. Tables 4.11 and 4.12 contain the running times achieved by MHyPro and Cora respectively, using the fastest strategies from the corresponding sets of strategies. The strategies used for the computations are listed in Tables 4.9 (MHyPro) and 4.10 (Cora). Due to the fact that the most strategies used smaller time-step sizes than for easy specification, Matlab's plotter had problems plotting the high number of segments. Therefore, here only the strategies will be compared.

Benchmark	Set Repr.	Time-Step Size [s]	Clust.
bb	box	5×10^{-5}	0
fo4	box	1×10^{-2}	0
fo8	box	2×10^{-2}	0
fo16	box	1×10^{-2}	0
sr	sf	1×10^{-2}	5
sw	sf	5×10^{-3}	5
rr	box	5×10^{-2}	0

Table 4.9: Fastest strategies for hard specification for MHyPro. Since no of the strategies could verify the two tanks and the vehicle platoon benchmarks, they are not listed in the table.

Benchmark	Time-Step Size [s]	Taylor Terms	Zonotope Order
bb	5×10^{-2}	2	2
fo4	1×10^{-3}	1	4
fo8	1×10^{-2}	1	200
sr	2×10^{-2}	100	200
sw	5×10^{-3}	100	200
rr	5×10^{-2}	2	2
tt	1×10^{-2}	40	10
vp	5×10^{-2}	100	200

Table 4.10: Fastest strategies for hard specification for Cora. Since no of the strategies could verify the benchmark filtered oscillator 16, it is not listed in the table.

For most of the benchmarks both tools needed more time than for easy specifications. Not surprising is the fact that MHyPro could not verify the two tanks and the vehicle platoon benchmarks and that Cora could not verify the filtered oscillator 16 benchmark. Both tools had already problems verifying this benchmarks with easy specifications.

First the results achieved by MHyPro are discussed. Except for the rod reactor benchmark, MHyPro needed clearly more time for the verification of these benchmarks in comparison to the times needed for the verification of the benchmarks with easy specifications. In most cases, the reason for the increased running times, is the fact

that the time-step size was decreased. An example is the bouncing ball benchmark. The strategy used for the verification of hard specifications used a one hundred times smaller time-step size than for the verification of the easy specifications. Therefore, it is not surprising the the reachability analysis algorithm needed about 75 times longer than for the reachability analysis with the strategy for the easy specifications. For the filtered oscillator 4 and 16, MHyPro needed about twice as long because in both cases the time-step size was decreased by half. The time-step size for the rod reactor benchmark was also decreased by half, however, the strategy needed nearly the same amount of time as for easy specifications. The reason for this computation time is the fact that for easy specification clustering was used but here, aggregation was used instead.

A clearly larger computation time was achieved for the spacecraft rendezvous benchmark. The strategy used for the verification of easy specifications needed 0.1812 s but the strategy used here needed 2.2985 s, i.e., about 13 times longer. The strategy is far more preciser then the one used for the verification of the easy specifications. First, instead of boxes the strategy used support functions as state set representation. Second, the time-step size was decreased by half, and third, the strategy used clustering instead of aggregation. All the changes explain the higher running time for this benchmark.

For the verification of the filtered oscillator 8 benchmark the strategy remained the same. However, the computation time is nearly twice as long as for the verification of easy specifications. The increased computation time is caused by the increased time for the verification. The verification algorithm needed about twice as long as for the easy specifications, however, the reachability analysis needed about the same amount of time.

For the switching system benchmark only the clustering was increased to 5. Therefore, the computation time is nearly the same as for the verification of the easy specifications.

Now the results obtained for Cora are discussed in detail. Similarly as for MHyPro, for most of the benchmarks, Cora needed more time for the verification. One exception is the bouncing ball benchmark. Here Cora needed nearly the same amount of time as for the verification of the easy specifications, although the strategies used for the computations differ in all parameters. The time-step size was decreased to one fourth of the time-step size used by the strategy for the easy specifications. The number of Taylor terms was decreased from 100 to 2 and the zonotope order was also decreased from 200 to 2. Still, Cora needed here only 1.055 longer. On the one hand the time-step size was decreased which increases the computation time but on the other hand the zonotope order as well as number of Taylor terms was decreased which decreases the computation time. Therefore, the computation times for both strategies are comparable.

For the filtered oscillator 4 and 8 benchmarks, Cora needed clearly more time. For the filtered oscillator 4 Cora needed about 16 times longer than for the verification of the easy specifications, and for the filtered oscillator 8 benchmark it needed about three times longer. The reason for the increased computation time for the filtered oscillator 4 benchmark is the fact that the time-step size was decreased by factor 50, while the number of Taylor terms and the zonotope order remained nearly the same. For the filtered oscillator 8 benchmark the reason for the increased computation time is the increased zonotope order (from 1 to 200). The decreased number of Taylor terms did not have that much impact on the computation time (from 100 to 1).

Not surprising is the result obtained for the spacecraft rendezvous benchmark.

CORA needed for the verification of the hard specifications about five times longer than for the verification of the easy specifications since the time-step size was decreased by factor 10. The result achieved for the rod reactor benchmark is similar.

For the switching system benchmark CORA needed about 1.3 times longer than for the verification of the easy specifications. The result is surprising since the time-step size was decreased by half and the number of Taylor terms and the zonotope order were increased by factor 10. This shows that increased zonotope order, or increased number of Taylor terms does not always imply increased computation time. This is especially true for the zonotope order since the parameter determines the maximal zonotope order, i.e., the zonotopes might have lower orders. The second fastest strategy for this benchmark used zonotope order of 400 and needed 1.1972 s for the reachability analysis. The strategy for easy specifications needed 1.2161 s for the reachability analysis whereby it used the same time-step size as the second fastest strategy for the hard specifications but the zonotope order was only 20.

Finally the results for the two tanks and the vehicle platoon are considered. The results obtained for the benchmarks are similar. For the vehicle platoon the same strategy was used as for the verification of the easy specifications. Therefore, the computation time is nearly the same. For the verification of the two tanks benchmark, the time-step size was decreased by factor 5 in comparison to the strategy used for the verification of the easy specifications. However, the decreased number of Taylor terms and the decreased zonotope order, avoided higher increase of the running time for this benchmark.

Benchmark	Reach. Time [s]	Verif. Time [s]	Running Time [s]
bb	2.2135	0.9050	3.1185
fo4	0.0346	0.2317	0.2663
fo8	0.0478	0.6115	0.6593
fo16	0.0433	126.4638	126.5071
sr	0.2827	2.0159	2.2985
sw	3.3512	9.5756	12.9268
rr	0.0671	0.2367	0.3039

Table 4.11: Running times obtained by MHYPRO for the hard specifications. The second column contains the times needed for the reachability analysis. The third column contains the times needed for the verification of the specifications. The last column contains the total times.

Benchmark	Reach. Time [s]	Verif. Time [s]	Running Time [s]
bb	1.2031	0.0736	1.2767
fo4	46.0048	3.9093	49.9141
fo8	23.7307	0.7284	24.4591
sr	1.7281	1.3629	3.0910
sw	1.5043	0.2961	1.8004
rr	3.6880	0.2851	3.9731
tt	1.5150	0.1110	1.6260
vp	4.5227	0.4179	4.9306

Table 4.12: Running times obtained by CORA for hard specifications. The second column contains the times needed for the reachability analysis. The third column contains the times needed for the verification of the specifications. The last column contains the total times.

4.2 Comparing Operations on Set Representations

Since HYPRO and CORA offer set representations not only for the reachability analysis, some of the remaining operations offered for the sets are considered now. Here only operations on zonotopes and boxes were compared since those are the only common set representations for MHYPRO and CORA. First the operations on zonotopes are considered.

CORA as well as HYPRO and MHYPRO, offer a number of various functions for zonotopes. In order to compare the performance of the operations twelve common functions were selected and tested in both tools. Those are:

- center (`center`) - returns the center of a zonotope.
- generators (`generators`)- returns the generators of a zonotope.
- Minkowski sum (`plus`) - returns the Minkowski sum of two zonotopes.
- unite (`unite`)- computes the union of two zonotopes.
- contains point (`containsPt`)- checks if a zonotope contains a given point.
- contains zonotope (`containsZono`)- checks if a zonotope contains an another zonotope.
- satisfies halfspaces (`satHalfspaces`) - checks if a zonotope satisfies a set of halfspaces of the form $C \cdot x \leq d, C \in \mathbb{R}^{m \times n}, d \in \mathbb{R}^m$.
- unite equal vectors (`deleteAligned`) - removes generators that are aligned.
- remove empty generators (`deleteZeros`) - removes generators whose entries are all zero.
- emptiness test (`empty`)- checks if a zonotope is empty.
- vertices (`vertices`) - returns vertices of a zonotope.

- reduce order (`reduceOrder`)- reduces the order of a zonotope.

The input for the functions were randomly generated zonotopes. For the Minkowski sum, union, vertices and the emptiness test, the zonotopes were defined by a randomly generated centers with 100 entries, where each entry had the value between 1000 and -1000. The generators were defined by randomly generated 100×10 matrices which entries also had the values in the range $[-1000, 1000]$. Note that the columns of the matrices represent the generators, not the rows. Therefore, for each generator g holds $g \in \mathbb{R}^100$. For the functions `center` and `generators`, the input was the zonotope obtained from the Minkowski sum. The zonotopes that were used for the `contains` functions, were generated by 3×3 matrices and 3×1 centers. For the function `deleteAligned` the generators were defined by a 4×3 matrix and a 4×1 center. For the order reduction function the generators were defined by a 4×10 matrix and a 4×1 center. In Table 4.13 the results are presented. The first two columns shows the running times for MHyPro and Cora, the third column contains results for HyPro. While the results for MHyPro and Cora are, in most cases, comparable, HyPro is clearly faster. Only for the function *contains point*, MHyPro is clearly faster than Cora.

Since here only single functions on set representations are considered, the overhead for MHyPro caused by Matlab has a greater impact on the run time than for example for the reachability analysis, where the most part of the computations are performed in C++.

Function	computation time [ms] MHyPro	Run Time [ms] Cora	Run Time [ms] HyPro
`center`	0.4030	0.3390	1.0000×10^{-3}
`generators`	0.2910	0.3860	1.0000×10^{-3}
`plus`	1.9190	5.6480	7.7000×10^{-2}
`unite`	2.7820	6.3960	6.7000×10^{-2}
`containsPt`	2.8230	575.8100	$<1.0000 \times 10^{-3}$
`containsZono`	2.2410	24.3990	3.0000×10^{-3}
`satHalfspaces`	124.9800	145.9100	2.2000×10^{-3}
`deleteAligned`	0.5310	15.3570	1.0000×10^{-3}
`deleteZeros`	0.4230	1.4710	$<1.0000 \times 10^{-3}$
`empty`	0.8500	1.7390	$<1.0000 \times 10^{-3}$
`vertices`	1.2250	440.3600	1.4630×10^{-2}
`reduceOrder`	0.6600	24.0780	4.0000×10^{-3}

Table 4.13: Performance comparison of HyPro, MHyPro and Cora on common zonotope functions.

The set representation *interval* in Cora is partially comparable with boxes offered by HyPro. Both representations have common functions but Cora offers some more functions that are more comparable with those offered by CArL for intervals, for example trigonometric functions, root function, power function, etc. The functions *plus*, *union*, and *vertices* compared in Table 4.14 are the same as for zonotopes. Additionally, the following functions were considered:

- *affine transformation* (`affineTrans`) - computes the affine transformation of

a box, i.e., a box is multiplied by a transformation matrix and shifted by a translation vector.

- *scale* (`scale`) - scales a box by a given factor.
- *supremum* (`sup`) - computes the supremum of a box.
- *intersect* (`intersect`) - computes the intersection of two boxes.

The results achieved by the tools are shown in Table 4.14. Again HyPro is clearly faster than Cora and MHyPro. A surprising result was achieved for the function computing vertices, as Cora needed about 1000 times longer than MHyPro. The same holds for the affine transformation. For Minkowski sum, however, MHyPro needed four times longer than Cora.

Function	Run Time [ms] MHyPro	Run Time [ms] Cora	Run Time [ms] HyPro
`affineTrans`	3.4480	25.9870	2.4600×10^{-1}
`intersect`	2.9650	4.3980	5.0000×10^{-3}
`plus`	17.7670	4.3440	1.1000×10^{-2}
`scale`	1.8970	1.9130	6.0000×10^{-3}
`sup`	1.0570	2.6590	7.0000×10^{-2}
`union`	7.2130	3.5750	7.0000×10^{-3}
`vertices`	1.0430	1112.7000	4.4000×10^{-2}

Table 4.14: Performance comparison of HyPro, MHyPro and Cora on common box functions.

As already mentioned, Cora provides some functions for intervals that are more comparable with those provided by CArL intervals. In order to compare the precision of CArL intervals that are used in HyPro, with intervals provided by Cora, a set of functions was considered. The functions are taken from [ZZZ10] and are shown in Table 4.15 together with the input values and exact solutions. The same functions were used by the Cora developers to compare the efficiency of Taylor terms implemented in Cora with those implemented in Flow* [AGK18]. The first four functions are uniform cubic B-splines that are commonly used for image warping applications [JLR03]. The remaining functions are multivariate polynomial functions.

Since all functions contain linear and non-linear operators (division and power operator) combined with different coefficients (positive, negative, large, small) and have different number of terms they are suitable for the observation of floating-point rounding errors. In order to see how precise the both tools are the error is computed as follows

$$\delta = \left[-\frac{\underline{x} - \underline{x}_{ref}}{\overline{x}_{ref} - \underline{x}_{ref}}, \frac{\overline{x} - \overline{x}_{ref}}{\overline{x}_{ref} - \underline{x}_{ref}} \right] \cdot 100\%$$

where $\underline{x}$ ($\overline{x}$) is the lower (upper) bound of the computed interval and $\underline{x}_{ref}$ ($\overline{x}_{ref}$) is the lower (upper) bound of the exact interval. If δ is equal zero then the calculated value is identical with the reference value. Large values of δ means large over-approximation. Analogous, very small values of δ means a great under-approximation [AGK18].

Table 4.16 contains the results achieved by Cora and HyPro. The results show that HyPro is clearly faster and slightly preciser than Cora. However, one has to

keep in mind that Cora currently does not pay attention to floating-point rounding errors like CArL, as Cora's focus lies on the development of reachability algorithms [AGK18]. HyPro could compute the upper bounds for the functions *image rejection*, *Mitchell*, and *three hump* more precisely than Cora. While the discrepancy for the *image rejection* function is not that big, for the other two functions the precision of Cora is clearly worse in comparison to HyPro. For function *three hump* also the calculation of the lower bound was more precise for HyPro than Cora.

Function	Initial Value	Accurate Range
bspline0		
$f_1(X) = \frac{1}{6} \cdot (1 - X)^3$	$X = [0, 1]$	$\left[0, \frac{1}{6}\right]$
bspline1		
$f_2(X) = \frac{1}{6} \cdot (3X^3 - 6X^2 + 4)$	$X = [0, 1]$	$\left[\frac{1}{6}, \frac{2}{3}\right]$
bspline2		
$f_3(X) = \frac{1}{6} \cdot (-3X^3 + 3X^2 + 3X + 1)$	$X = [0, 1]$	$\left[\frac{1}{6}, \frac{2}{3}\right]$
bspline3		
$f_4(X) = \frac{1}{6} \cdot (-X^3)$	$X = [0, 1]$	$\left[-\frac{1}{6}, 0\right]$
Savitzky-Golay filter		
$f_5(X) = 7x_1^3 - 984x_2^3 - 76x_1^2x_2 + 92x_1x_2^2 + 7x_1^2 - 39x_1x_2 - 46x_2^2 + 7x_1 - 46x_2 - 75$	$X = [-2, 2]^2$	$[-9.453, 9.303]$
Image Rejection Unit		
$f_6(X) = 16384(x_1^4 + x_2^4) + 64767(x_1^2 - x_2^2) + x_1 - x_2 + 57344x_1x_2(x_1 - x_2)$	$X = [0, 1]^2$	$\left[-5.51 \times 10^4, 8.79 \times 10^4\right]$
A Random Function		
$f_7(X) = (x_1 - 1)(x_1 + 2)(x_2 + 1)(x_2 - 2)x_3^2$	$X = [-2,2]^3$	$[-36, 64]$
Mitchell Function		
$f_8(X) = 4[x_1^4 + (x_2^2 + x_3^2)^2] + 17x_1^2(x_2^2 + x_3^2) - 20(x_1^2 + x_2^2 + x_3^2) + 17$	$X = [-2,2]^3$	$[-8, 641]$
Matyas Function		
$f_9(X) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$	$X = [-100,100]^2$	$[0, 10\,000]$
Three Hump Function		
$f_{10}(X) = 12x_1^2 - 6.3x_1^4 + x_1^6 + 6x_2(x_2 - x_1)$	$X = [-10,10]^2$	$\left[0, 0.94 \times 10^6\right]$
Ratscheck Function		
$f_{11}(X) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$	$X = [-100,100]^2$	$\left[-1.03, 0.33 \times 10^{12}\right]$

Table 4.15: Functions used for the comparison of operations on intervals for HyPro and Cora.

	HyPro			Cora		
Function	Result	Error	computation time [ms]	Result	Error	computation time [ms]
$f_1(X)$	[0.00, 0.17]	[0.00, 0.00]	6.00×10^{-3}	[0.00, 0.17]	[0.00,0.00]	2.50
$f_2(X)$	[−0.33, 1.17]	[100.00, 100.00]	7.00×10^{-3}	[−0.33, 1.17]	[100.00,100.00]	3.93
$f_3(X)$	[−0.17, 0.00]	[100.00, 100.00]	6.00×10^{-3}	[−0.33, 1.17]	[100.00,100.00]	5.22
$f_4(X)$	[−0.17, 0.00]	[0.00, 0.00]	2.00×10^{-3}	[−0.17, 0.00]	[0.00,0.00]	0.59
$f_5(X)$	[−9793.00, 9487.00]	[52 162.20, 50 531.50]	1.40×10^{-2}	[−9793.00, 9487.00]	[52 162.23, 50 531.55]	16.19
$f_6(X)$	[−122 112.00, 138 497.00]	[46.86, 35.38]	8.00×10^{-3}	[−122 112.00, 154 880.00]	[46.86, 46.84]	2.15
$f_7(X)$	[−192.00, 576.00]	[156.00, 512.00]	8.00×10^{-3}	[−192.00, 576.00]	[156.00, 512.00]	2.79
$f_8(X)$	[−223.00, 881.00]	[33.13, 36.98]	9.00×10^{-3}	[−223.00, 2225.00]	[33.13, 244.07]	3.84
$f_9(X)$	[−4800.00, 10 000.00]	[48.00, 1.82×10^{-14}]	3.00×10^{-3}	[−4800.00, 10 000.00]	[48.00, 0.00]	3.48
$f_{10}(X)$	[−64 200.00, 1.00×10^{6}]	[6.83, 6.64]	5.00×10^{-3}	[−1 064 200.00, 2400.00]	[113.21, −99.74]	2.74
$f_{11}(X)$	[-2.10×10^{8}, 3.34×10^{11}]	[0.06, 1.13]	6.00×10^{-3}	[-2.10×10^{8}, 3.34×10^{11}]	[0.06, 1.13]	4.94

Table 4.16: Results obtained by HyPro and Cora for functions taken from [ZZZ10].

4.3 Reachability Analysis in HyPro and MHyPro

In the previous section set operations for HyPro and MHyPro were compared. In this section, the overhead for the reachability analysis for MHyPro is shortly considered. In order to estimate the overhead caused by Matlab for MHyPro, the benchmarks used previously for the comparison with Cora, were used here. Additionally, the results were compared with those achieved by HyDra, which is a tool that uses the data types offered by HyPro to implement various reachability analysis algorithms for hybrid systems.

Table 4.17 contains the results obtained for the three tools. For each benchmark, the tools used the same strategy. For the benchmarks which names contain the abbreviation 'init' the initial settings were used. For the benchmarks which names contain the abbreviation 'easy' ('hard') the strategies for the verification of the easy (hard) specifications were used.

For all benchmarks with safety specifications, HyPro was always the fastest tool. MHyPro is faster than HyDra on 5 out of 13 benchmarks with safety specifications. On average ,for this class of benchmarks, MHyPro needed about 400 times longer than HyPro and 30 times longer than HyDra. The reason for the overhead is the verification algorithm that is implemented in Matlab. This can be acknowledged when only the benchmarks without safety specifications are considered. For three out of nine benchmarks MHyPro is faster than HyPro and for six benchmarks faster than HyDra. On average, for this class of benchmarks, HyPro is only 1.17 times faster than MHyPro and HyDra is about 2.58 times slower than MHyPro.

Benchmark	Time [s] MHyPro	computation time [s] HyDra	computation time [s] HyPro
bb_init	0.0214	0.207 894	8.655×10^{-3}
bb_easy	0.1280	0.340 912	9.379×10^{-3}
bb_hard	3.1185	23.4108	5.639×10^{-3}
fo4_init	0.0063	0.153 583	8.999×10^{-3}
fo4_easy	0.1488	0.215 151 79	9.608×10^{-3}
fo4_hard	0.2663	0.117 016	7.913×10^{-3}
fo8_init	0.0046	0.161 662	1.2807×10^{-2}
fo8_easy	0.3697	0.235 144	1.3899×10^{-2}
fo8_hard	0.6593	0.355 57	9.67×10^{-3}
fo16_init	0.0071	0.190 213	2.2501e-2
fo16_easy	64.9006	0.297 212	2.1891e-2
rr_init	0.1265	1.355 66	6.9417×10^{-2}
rr_easy	0.2932	0.101 323	-
rr_hard	0.3039	4.542 74	7.4183×10^{-2}
sr_init	0.0577	0.833 885	3.8066×10^{-2}
sr_easy	0.1812	0.519 526	4.1803×10^{-2}
sr_hard	2.2985	1.491 02	-
sw_init	25.6838	0.588 405	20.1933
sw_easy	12.7854	0.174 179	-
sw_hard	12.9268	0.171 552	-
tt_init	0.0091	0.106 43	8.355×10^{-3}
vp_init	9.1345	0.492 808	8.4431

Table 4.17: Comparison of the running times obtained by HyPro, HyDra, and HyPro. For each benchmark the initial strategy and the strategies for easy and hard specifications were considered. The benchmark names with ending *'_init'* signify usage of the initial strategy. The ending *'_easy'* (*'_hard'*) indicates that the strategy for easy (hard) specification was considered.

Chapter 5

Conclusion

This thesis proposes a MATLAB-wrapper for the C++ tool HYPRO offering various state set representations and algorithms for analysis of hybrid systems. First a short introduction to hybrid systems and the verification methods was provided. Especially the reachability algorithm and state set representations were explained. The core of this thesis is the comparison of HYPRO with the well known tool CORA whereby the wrapper for HYPRO, called MHYPRO, was used. The efficiency and precision of MHYPRO, HYPRO, and CORA was evaluated using a set of nine well know benchmarks with two different sets of safety specifications. Besides the efficiency of the reachability algorithms, the efficiency of single state set operations for zonotopes and boxes was compared. Since CORA intervals offer functions that are more comparable with those offered by CArL intervals, that are used by HYPRO, the precision of some interval functions was also evaluated. Finally, the overhead caused by MATLAB for MHYPRO was analysed.

The comparison of HYPRO and CORA showed that both tools have advantages and disadvantages. CORA offers reachability analysis algorithms for continuous, discrete, and hybrid dynamics. It supports linear and nonlinear systems. The high number of parameters, makes it possible for the user to customize the models more precisely than with HYPRO. However, the impact of the particular parameters on the analysis results is not well documented and often difficult to understand. Even though in scope of this thesis only few parameters were considered and analyzed, it took time to understand what impact the parameters have on the results, and what default values can be taken as initial setting. However, as the vehicle platoon benchmark has shown, it is sill not always clear how the parameters have to be set.

HYPRO is less complex than CORA, however, it does not provide that much modeling possibilities in comparison to CORA. Important features like parallel composition and support of algebraic variables need to be in integrated in order to make the tool more attractive for the industry since those are features that are constantly needed in the practice. The development of the MATLAB wrapper makes HYPRO more interesting for engineers using MATLAB. The fact that the performance of MHYPRO is comparable with the performance of HYPRO could make the wrapper competitive when further features of HYPRO, that are still missing, are integrated.

5.1 Future Work

There are various possibilities to continue this work. The possibilities concern the implementation of the wrapper and HyPro, as well as the comparison with Cora. First further possibilities for the comparison will be shortly discussed. The comparison made in scope of this thesis concerned only nine benchmarks. However, it would be interesting how well MHyPro (and HyPro) performs on some other benchmark from the ARCH competition, since many of the benchmarks come from the real world and are challenging.

There is a number of possibilities to improve MHyPro. An important improvement would be the implementation of a parser for Flow* or SpaceEx syntax since currently the modelling process is complex and thus error-prone. Another possibility would be the implementation of a graphical user interface. Matlab provides a tool called *App Designer* that could be one possible way to do this. Moreover, the remaining state set representations could be implemented, e.g., Taylor terms, and some data structures could be wrapped that are still missing.

In general, a possible next step for HyPro would be the implementation of further dynamics, since many problems are nonlinear. Moreover, the possibility of adding external input should be enabled and the parallel composition should be implemented. This improvements would make HyPro even more comparable with tools like Cora.

Bibliography

[Ábr12] E. Ábrahám. Modeling and analysis of hybrid systems. *RWTH Aachen University, Lecture Notes*, 2012.

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

[AGK18] M. Althoff, D. Grebenyuk, and N. Kochdumper. Implementation of Taylor models in CORA 2018. In *EPiC Series in Computing*, volume 54, pages 115–145. EasyChair, 2018.

[AK11] M. Althoff and B. Krogh. Zonotope bundles for the efficient computation of reachable sets. In *Proceedings of the IEEE Conference on Decision and Control*, pages 6814–6821, 2011.

[AKA18] M. Althoff, N. Kochdumper, and C. Arch. CORA 2018 manual. Technical report, Technische Universitaet Muenchen, 2018.

[Alt10] M. Althoff. *Reachability analysis and its application to the safety assessment of autonomous cars.* Dissertation, Technische Universitaet Muenchen", Muenchen, 2010.

[Alt13] M. Althoff. Reachability analysis of nonlinear systems using conservative polynomialization and non-convex sets. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, page 173. ACM Press, 2013.

[Alu11] R. Alur. Formal verification of hybrid systems. In *Proceedings of the Ninth ACM International Conference on Embedded Software*, pages 273–278. IEEE, 2011.

[ASB07] M. Althoff, O. Stursberg, and M. Buss. Reachability analysis of linear systems with uncertain parameters and inputs. In *2007 46th IEEE Conference on Decision and Control*, pages 726–732. IEEE, 2007.

[ASB09] M. Althoff, O. Stursberg, and M. Buss. Safety assessment for stochastic linear systems using enclosing hulls of probability density functions. In *2009 European Control Conference (ECC)*, pages 625–630. IEEE, 8 2009.

[BK08] C. Baier and J.-P. Katoen. *Principles of model checking.* The MIT press, 2008.

[CÁS13] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[CM17] N. Chan and S. Mitra. Verifying safety of an autonomous spacecraft rendezvous mission. *arXiv preprint arXiv:1703.06930*, 48:20–6, 2017.

[CPPAV06] L. P. Carloni, R. Passerone, A. Pinto, and A. L. Angiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends® in Electronic Design Automation*, 1:1–193, 2006.

[dAe15] Verband der Automobilindustrie eV. Automation: From driver assistance systems to automated driving. *VDA Magazine-Automation*, page 2, 2015.

[DF13] A. Donze and G. Frehse. Modular, hierarchical models of control systems in spaceex. In *2013 European Control Conference*, pages 4244–4251. IEEE, 2013.

[DLV11] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling cyber–physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2011.

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

[Flo93] R. W Floyd. Assigning meanings to programs. In *Program Verification*, pages 65–81. Springer, 1993.

[Gar85] C. W. Gardiner. *Handbook of stochastic methods for physics, chemistry and the natural sciences*. Springer, 1985.

[GH04] A. R. Girard and A. S. Howell. Model-driven hybrid and embedded software for automotive applications. In *Second RTAS Workshop on Model-Driven Embedded Systems*, pages 25–28. Citeseer, 2004.

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

[Hen00] Thomas A Henzinger. The theory of hybrid automata. In *Verification of digital and hybrid systems*, pages 265–292. Springer, 2000.

[HKPV98] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[JLR03] J. Jiang, W. Luk, and D. Rueckert. Fpga-based computation of free-form deformations in medical image registration. In *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology*, pages 234–241. IEEE, 2003.

[KSA17] A. K. Kopetzki, B. Schurmann, and M. Althoff. Methods for order reduction of zonotopes. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 5626–5633. IEEE, 2017.

[Lee10] E. A Lee. CPS foundations. In *Proceedings of the 47th Design Automation Conference on - DAC '10*, pages 737–742. IEEE, 2010.

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

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1:134–152, 1997.

[LS16] E. A. Lee and S. A. Seshia. *Introduction to embedded systems. A cyber-physical systems approach.* The MIT Press, 2nd edition, 2016.

[Mat] Mathworks. C MEX API – MATLAB. Accessed: 2019-05-13.

[MKC09] R. E. Moore, R. B. Kearfott, and M. J. Cloud. *Introduction to Interval Analysis.* Society for Industrial and Applied Mathematics, 2009.

[SÁMK17] S. Schupp, E. Ábrahám, I. B. Makhlouf, and S. Kowalewski. Hypro: A c++ library of state set representations for hybrid systems reachability analysis. In *NASA Formal Methods Symposium*, pages 288–294. Springer, 2017.

[SCN13] R. Sanfelice, D. Copp, and P. Nanez. A toolbox for simulation of hybrid systems in matlab/simulink: Hybrid equations (hyeq) toolbox. In *Proceedings of the 16th international conference on Hybrid systems: computation and control*, pages 101–106. ACM, 2013.

[SRMB16] J. K. Scott, D. M. Raimondo, G. R. Marseglia, and R. D. Braatz. Constrained zonotopes: A new tool for set-based estimation and fault detection. *Automatica*, 69:126–136, 2016.

[Zie14] G. M. Ziegler. *Lectures on polytopes.* Springer, 2014.

[ZZZ10] L. Zhang, Y. Zhang, and W. Zhou. Tradeoff between approximation accuracy and complexity for range analysis using affine arithmetic. *Journal of Signal Processing Systems*, 2010.

Appendix A

Appendix

A.1 Benchmarks Models

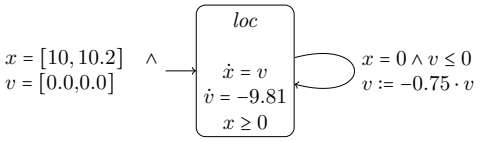


Figure A.1: Hybrid automaton modelling the bouncing ball benchmark.

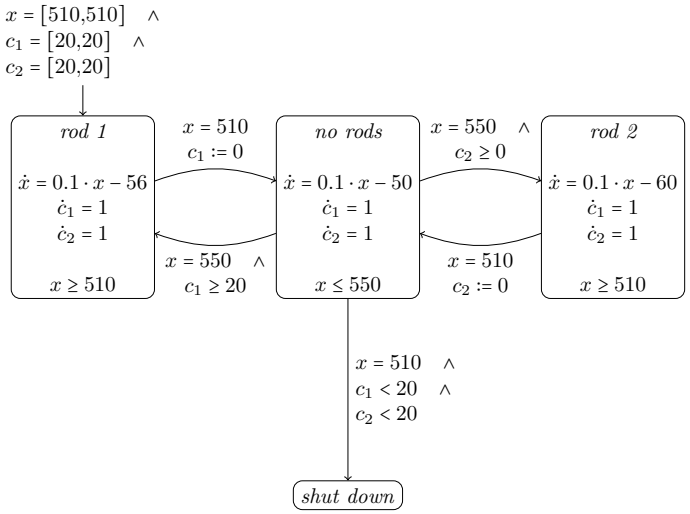


Figure A.2: Hybrid automaton modelling the rod reactor benchmark.

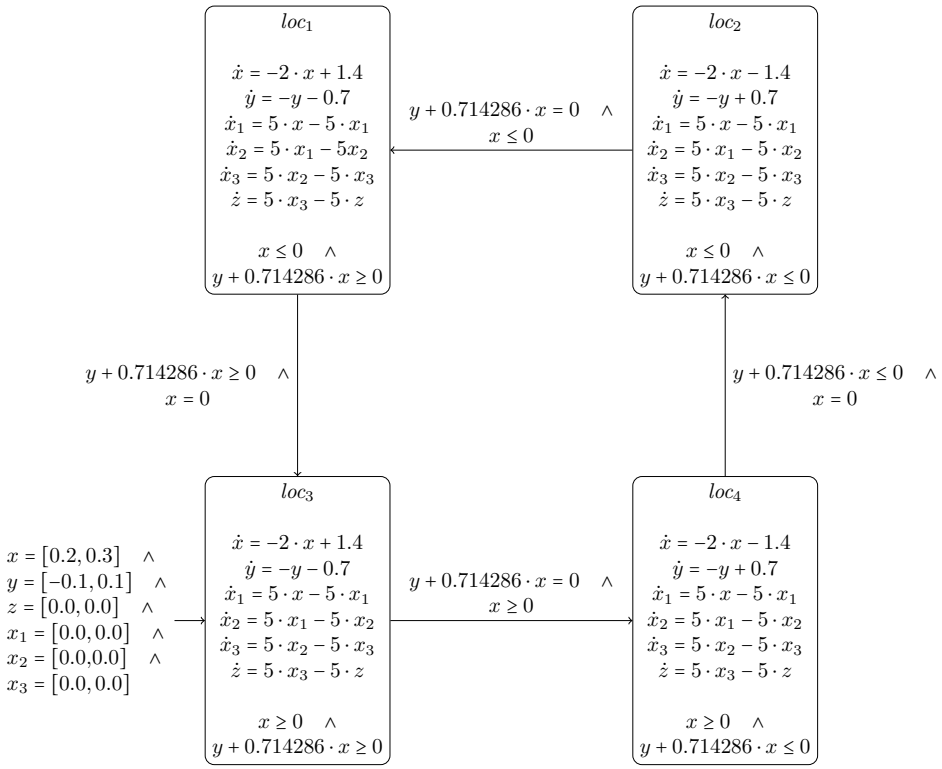


Figure A.3: Hybrid automaton modelling the filtered oscillator benchmark.

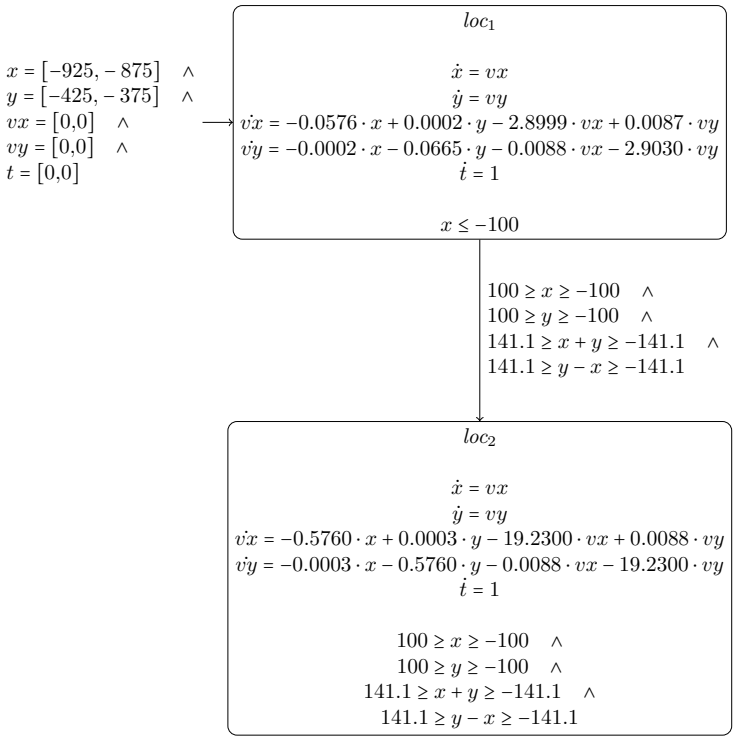


Figure A.4: Hybrid automaton modelling the spacecraft rendezvous benchmark.

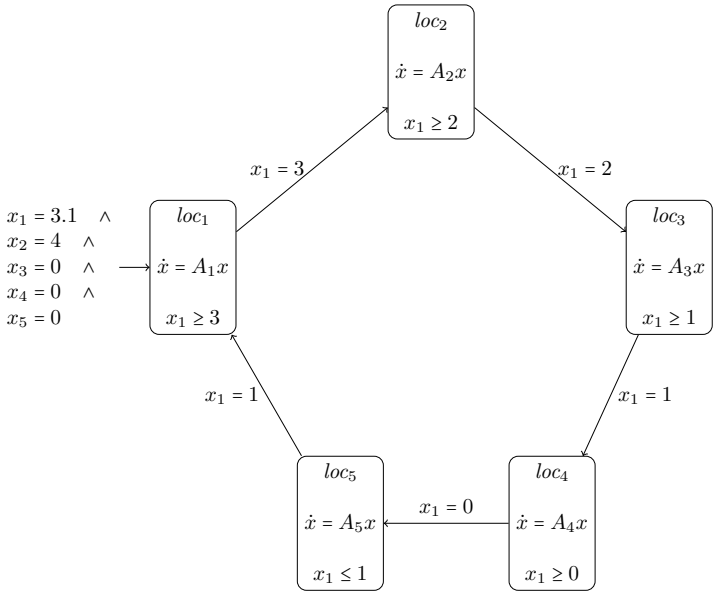


Figure A.5: Hybrid automaton modelling the switching system benchmark.

$$A_1 = \begin{pmatrix} -0.8047 & 8.7420 & -2.4591 & -8.2714 & -1.8640 \\ -8.6329 & -0.5860 & -2.1006 & 3.6035 & -1.84203 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 1.8302 & 1.9869 & -2.4539 & -1.7726 & -0.7911 \end{pmatrix}$$

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

$$A_3 = \begin{pmatrix} -0.9275 & 8.8628 & 2.5428 & -8.2329 & -2.0324 \\ -0.8316 & -0.5860 & 2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 0.7635 & 3.0357 & -3.1814 & -1.4388 & -2.2538 \end{pmatrix}$$

$$A_4 = \begin{pmatrix} -1.0145 & 8.9701 & -2.6207 & -8.2199 & -2.1469 \\ -0.8316 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ 0.0076 & 3.9682 & -3.8578 & -1.3253 & -3.2477 \end{pmatrix}$$

$$A_5 = \begin{pmatrix} -1.4021 & 10.1647 & -3.3937 & -8.5139 & -2.9602 \\ -0.8316 & -0.5860 & -2.1006 & 3.6035 & -1.8423 \\ 2.4511 & 2.2394 & -0.7538 & -3.6934 & 2.4585 \\ 8.3858 & -3.1739 & 3.7822 & -0.6249 & 1.8829 \\ -3.3585 & 14.3426 & -10.5703 & -3.8785 & -10.3111 \end{pmatrix}$$

$$x = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 \end{pmatrix}^T$$

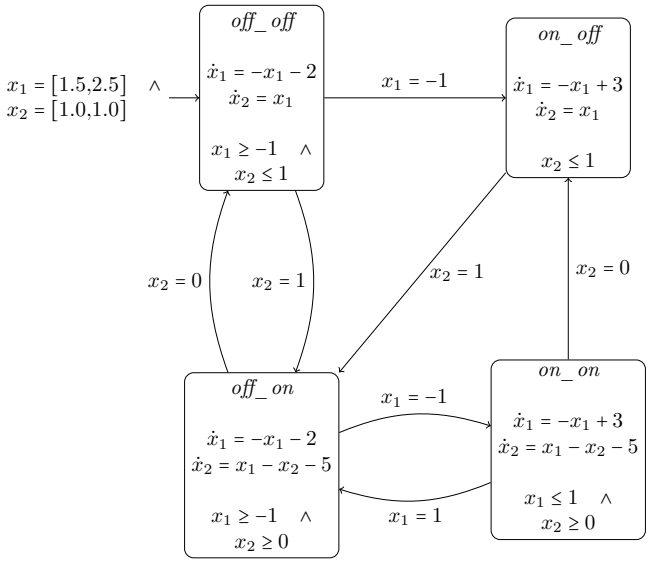


Figure A.6: Hybrid automaton modelling the two tanks benchmark.

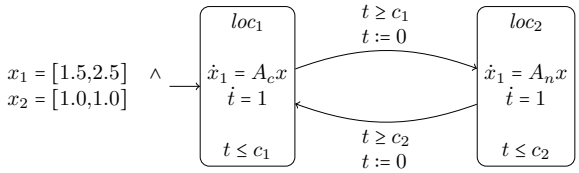


Figure A.7: Hybrid automaton modelling the vehicle platoon benchmark.

$$A_c = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.6050 & 4.8680 & -3.5754 & -0.8198 & 0.4270 & -0.0450 & -0.1942 & 0.3626 & -0.0946 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0.8718 & 3.8140 & -0.0754 & 1.1936 & 3.6258 & -3.2396 & -0.5950 & 0.1294 & -0.0796 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0.7132 & 3.5730 & -0.0964 & 0.8473 & 3.2568 & -0.0876 & 1.2726 & 3.0720 & -3.1356 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$A_n = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1.6050 & 4.8689 & -3.5754 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.1936 & 3.6258 & -3.2396 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0.7132 & 3.5730 & -0.0964 & 0.8472 & 3.2568 & -0.0876 & 1.2726 & 3.0720 & -3.1356 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$$x = \begin{pmatrix} e_1 & \dot{e}_1 & a_1 & e_2 & \dot{e}_2 & a_2 & e_3 & \dot{e}_3 & a_3 & t \end{pmatrix}^T$$

A.2 Strategies Used for the Evaluation

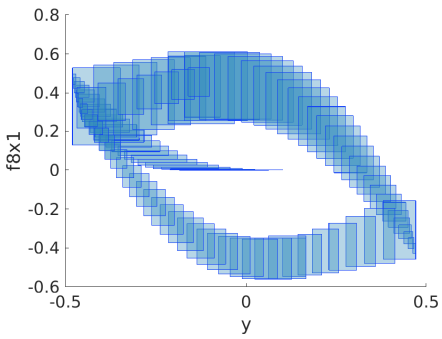
Strategy Nr.	Time Step Size [s]	Taylor Terms	Zonotope Order
1	1×10^{-1}	100	200
2	1×10^{-2}	100	200
3	1×10^{-3}	100	200
4	1×10^{-3}	50	100
5	1×10^{-1}	50	100
6	1×10^{-1}	10	20
7	2×10^{-2}	1	5
8	1×10^{-2}	1	1
9	5×10^{-3}	100	200
10	1×10^{-2}	10	20
11	5×10^{-1}	10	20
12	1×10^{-2}	25	50
13	1×10^{-1}	25	50
14	1×10^{-2}	1	200
15	1×10^{-2}	100	1
16	2×10^{-2}	100	200
17	2×10^{-1}	100	200
18	1×10^{-2}	10	40
19	1×10^{-2}	40	10
20	2×10^{-4}	100	200
21	2×10^{-4}	5	200
22	5×10^{-5}	100	200
23	5×10^{-5}	10	400
24	1×10^{-3}	5	400
25	1×10^{-1}	5	400
26	1×10^{-2}	5	400
27	2×10^{-2}	100	200
28	2×10^{-2}	200	300
29	1×10^{-2}	200	300
30	1×10^{-3}	200	300
31	1×10^{-1}	200	300
32	5×10^{-2}	100	200
33	1×10^{-3}	1	4
34	5×10^{-2}	2	2
35	1×10^{-2}	1	2
36	5×10^{-3}	100	200

Table A.1: Settings for CORA that were used for the evaluation of the benchmarks.

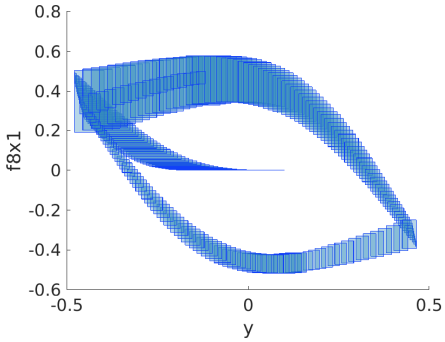
Strategy Nr.	Time-Step Size [s]	State Set Representation	Clust.
1	1×10^{-2}	box	5
2	1×10^{-2}	box	3
3	1×10^{-2}	box	0
4	1×10^{-1}	box	5
5	1×10^{-1}	box	3
6	1×10^{-1}	box	0
7	5×10^{-2}	box	5
8	5×10^{-2}	box	3
9	5×10^{-2}	box	0
10	2×10^{-2}	box	5
11	2×10^{-2}	box	3
12	2×10^{-2}	box	0
13	5×10^{-5}	box	5
14	5×10^{-5}	box	3
15	5×10^{-5}	box	0
16	1×10^{-3}	box	5
17	1×10^{-3}	box	3
18	1×10^{-3}	box	0
19	5×10^{-3}	box	5
20	5×10^{-3}	box	3
21	5×10^{-3}	box	0
22	1×10^{-2}	sf	5
23	1×10^{-2}	sf	3
24	1×10^{-2}	sf	0
25	1×10^{-1}	sf	5
26	1×10^{-1}	sf	3
27	1×10^{-1}	sf	0
28	5×10^{-2}	sf	5
29	5×10^{-2}	sf	3
30	5×10^{-2}	sf	0
31	2×10^{-2}	sf	5
32	2×10^{-2}	sf	3
33	2×10^{-2}	sf	0
34	5×10^{-5}	sf	5
35	5×10^{-5}	sf	3
36	5×10^{-5}	sf	0
37	1×10^{-3}	sf	5
38	1×10^{-3}	sf	3
39	1×10^{-3}	sf	0
40	5×10^{-3}	sf	5
41	5×10^{-3}	sf	3
42	5×10^{-3}	sf	0

Table A.2: Settings for MHyPro that were used for the evaluation of the benchmarks. The abbreviation *Clust.* stands for clustering. stands for aggregation. If the value for clustering is set to 0 it means that clustering is not used and instead aggregation is used.

A.3 Flowpipes Computed for Some Benchmarks

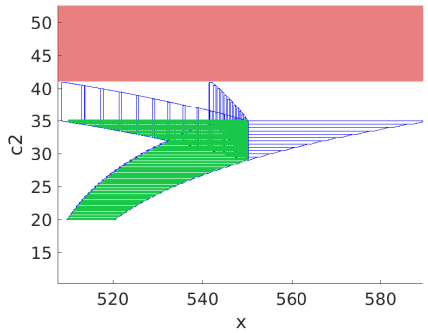


(a) Reachability analysis of the filtered oscillator 16 benchmark computed by MHyPro with the initial settings.

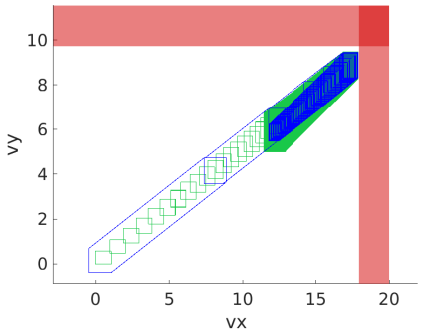


(b) Reachability analysis of the filtered oscillator 16 benchmark computed by MHyPro with strategy for the easy specifications.

Figure A.8: Flowpipes computed by MHyPro for the filtered oscillator benchmark 16 with the initial settings (left) and strategy for the easy specifications (right).

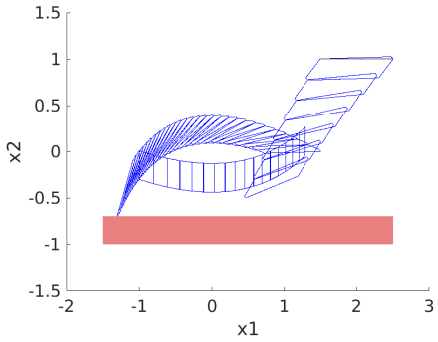


(a) Reachability analysis of the rod reactor benchmark performed by Cora (blue) and MHyPro (green) with strategy for the easy specification.

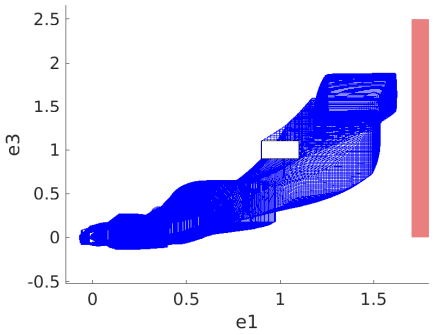


(b) Reachability analysis of the spacecraft rendezvous benchmark performed by Cora (blue) and MHyPro (green) with strategy for the easy specification.

Figure A.9: Flowpipes computed by MHyPro (green) and Cora (blue) for the rod reactor benchmark (left) and spacecraft rendezvous benchmark (right) with strategies for the easy specifications. The red rectangles depict the bad states.



(a) Reachability analysis of the two tanks benchmark performed by CORA with the strategy for the easy specifications.



(b) Reachability analysis of the vehicle platoon benchmark performed with strategy for the easy specification.

Figure A.10: Flowpipes computed by CORA for the two tanks benchmark (left) and vehicle platoon benchmark (right) with strategies for easy specifications. The red rectangles depict the bad states.