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

CONTEXT-DEPENDENT REACHABILITY ANALYSIS FOR HYBRID AUTOMATA

Justin Winkens

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

Additional Advisor:
Stefan Schupp M.Sc.

Aachen, May 22, 2018

Abstract

This thesis is focused on the analysis of *hybrid automata*, which incorporate discrete and continuous behavior of a given system into one model.

A common approach to analyze these automata for safety is so called *flowpipe construction* that iteratively constructs a set of over-approximative geometric objects that contain all reachable states of the automaton. Implementations of this approach can be found in toolboxes like Flow*, HyPro/HyDRA or SpaceEx.

Unfortunately, the computational cost of flowpipe construction is exponential in the number of variables/dimensions.

This thesis aims at improving the HyPro/HyDRA toolbox by implementing a context-based approach to flowpipe analysis using variable set separation, in which the state space of the underlying automaton is decomposed into syntactically independent sets of variables allowing for separate analysis of the now lower dimensional subspaces.

The different subspaces can then be analyzed for their properties enabling us to perform a *context-based analysis* where we will be able to dynamically choose data structures and algorithms based on the context defined by the automaton's current location and the state's subspaces.

In the process HyPro is augmented with a new representation called *difference bound matrix* that is especially suited for the analysis of timed automata.

Experimental results in common benchmarks show a speedup of up to 11 000% for decomposed analysis. For decomposed analysis with context-based optimization a speedup of up to 200 000% can be observed.

Contents

Chapter 1

Introduction

In our everyday lives we are surrounded by *hybrid systems* that exhibit *discrete* and *continuous* behavior. Upon waking up you raise the thermostat, brew a cup of coffee or tea, fry an egg for breakfast and then drive to work in your car. All these activities involve the use of hybrid systems.

The thermostat continuously changes the temperature after the discrete step of raising it while the coffee machine continuously produces coffee after being turned on. The stove continuously produces heat after being turned on and your car is controlled by dozens of microcontrollers. The digital systems controlling these devices interact with our continuous world and together make up a hybrid system.

Hybrid systems are everywhere and we use them without even thinking, relying on their functionality and safety. We trust in the manufacturer who hopefully tested the ins and outs of their product. But all this testing will never guarantee safety. Theres always a chance that the testers stopped their tests too early, maybe even one iteration before a critical malfunction.

The past has been riddled with stories of car manufactures calling back millions of vehicles due to malfunction of various hybrid components such as cruise control, resulting in e.g. uncontrollable acceleration.

All these concerns raised the needs for more formal methods and spawned the research area of *hybrid systems verification* trying to ensure functionality and safety.

In verification of hybrid systems we try to confirm that a system has certain properties. One of these properties is the so called *safety* property which states that nothing bad is going to happen with the system, i.e. no critical malfunctions occur. This malfunctioning is often characterized by a set of bad states. Given a system and an initial state we then try to show that these bad states cannot occur and ipso facto the system is safe.

Unfortunately, proving that a hybrid system is safe is undecidable but by using *over-approximative* techniques we can at least come up with *semi-decision procedures*.

One of these semi-decision procedures to show that a system is safe and therefore does not reach its bad states is by taking its initial state and over-approximatively compute reachable successor states ultimately yielding a set of states that hopefully does not contain the bad states. Since the over-approximation of reachable successor states does not contain the bad states, we know that the actual successor states do not contain the bad states as well. However, if the over-approximation contains the bad states, we can not say whether or not the system is unsafe because the bad states

may only be contained in the over-approximation and not the actual successor states.

This semi-decision procedure is called *flowpipe construction* and there are many toolboxes implementing this approach such as Flow* [CÁS13], SpaceEx [FLGD+11] and HyPro/HyDRA [SÁMK17]. These tools differ in the used techniques and the way they represent the state of a system.

In this thesis we will improve the HyPro/HyDRA toolbox by implementing a divide and conquer approach to verification of hybrid systems based on *variable set separation*.

Furthermore, the toolbox will be augmented by a new representation and analysis algorithm for timed systems (an important subclass of hybrid systems) based on *zones*.

The main contribution of this thesis is a strategy that is able to choose the most suitable representation and technique for the different variable sets based on the current *context* of the system to significantly improve the performance of the HyPro/HyDRA toolbox.

Outline In Chapter 2 we are going to introduce all necessary preliminaries needed to understand what hybrid systems are and how to analyze them using flowpipe construction. Additionally, timed automata are explored and the concept of zone-based analysis is introduced. Subsequently, in Chapter 3 we will learn about the HyPro toolbox and augment it with a new data structure called zones. Thereafter, in Chapter 4 we explore the HyDRA toolbox and enhance it with a framework for analysis based on variable set separation and an algorithm for analyzing timed automata. Finally, in Chapter 5 we are going to evaluate our implementations against common benchmarks.

1.1 Related Work

This thesis combines the work of multiple researchers into one powerful analysis algorithm. The idea of a divide and conquer approach to hybrid systems analysis has been explored in [SNÁ17] and [CS16]. This idea is combined with software design ideas from [FR09] to implement a variable set separation-based analysis algorithm in the HyPro/HyDRA toolbox, in which algorithms and operation implementations can be switched dynamically.

The representation and analysis algorithm for timed systems is based on the work of [BY04] and [Sri12]. In this thesis, we will elaborate on the idea of difference bound matrices as a representation by defining additional operations which makes them suitable to be used in a flowpipe construction algorithm instead of a zone-based abstraction algorithm.

Chapter 2

Hybrid Systems Analysis

In this thesis we are concerned with the reachability problem in hybrid systems. Our goal is to find out if the system at hand can reach a (possibly undesirable) state. This chapter introduces the necessary theoretical background to perform reachability analysis for hybrid systems.

We will start by defining an abstraction for hybrid systems by using hybrid automata. Then, we will define the semantics of such an automaton followed by a description of the reachability problem. Subsequently, we will investigate the common analysis approach of flowpipe construction for linear hybrid automata, an important subclass of hybrid automata. Finally, we will introduce the concept of timed automata. We will see that for this subclass the reachability problem is easier to solve than for general linear hybrid automata.

2.1 Hybrid Automata

Hybrid systems describe both continuous and discrete behavior and a common approach to model these systems is by means of *hybrid automata*. A hybrid automaton's continuous behavior is described by *activities* in their *locations* that define the evolution of variables over a certain amount of time. The discrete behavior is modeled by *transitions* that put the automaton from one *state* into another. Formally, a hybrid automaton can be given by the following definition:

Definition 2.1.1 (Hybrid automaton [ACHH93])
A hybrid automaton A is described by a tuple $A = (Loc, Var, Lab, Edge, Act, Inv, Init)$ where

- *Loc is a finite set of locations.*
- *Var is a finite set of real-valued variables. A function $v : Var \to \mathbb{R}$ that assigns values to variables is called valuation. The set of valuations of all variables is denoted by $\mathcal{V}$.*
- *Lab is a finite set of (synchronization) labels.*
- *$Edge \subseteq Loc \times Lab \times 2^{\mathcal{V} \times \mathcal{V}} \times Loc$ is a finite set of edges or transitions. Transitions are described by a transition relation $\mu \in \mathcal{V} \times \mathcal{V}$ that specifies guard and effect of*

a transition. We can take a transition with a valuation v changing the valuation to v' iff $(v,v') \in \mu$.

- *Act is a function that assigns a set of time invariant activities $f : \mathbb{R}_{\geq 0} \to \mathcal{V}$ to each location. These activities model the continuous dynamics of a variable in a location.*

- *Inv is a function that assigns a set of invariants $Inv(l) \subseteq \mathcal{V}$ to each location that specifies the set of valid valuations for a location. This enforces discrete behavior as the automaton must leave a location before the valuation violates the location's invariant.*

- *$Init \subseteq Loc \times \mathcal{V}$ is a set of initial states. The set of all states is denoted as Σ.*

Note that in later chapters when things become more technical we refer to *activities* as *flows*. A common example for a hybrid system is a thermostat.

Example 2.1.1 (Thermostat [ACHH93])
A thermostat is a controller that tries to maintain a certain temperature in a room. In order to achieve this goal it senses the temperature in the room and if it is lower (higher resp.) than the desired temperature it turns the heater on (off resp.).

In this example the controller tries to maintain a temperature between $18°C$ and $22°C$. Initially we assume a temperature of $x = 20°C$ and that the heater is turned on. If the heater is turned on the continuous dynamics of the room temperature can be given by the differential equation $\dot{x} = K(h - x)$ where K and h are some physical constants.

If the heater is turned off the continuous dynamics of the room temperature can be given by the differential equation $\dot{x} = -Kx$. If the temperature reaches $22°C$ the controller turns off the heating and turns it back on if the temperature has fallen to $18°C$.

Considering Example 2.1.1 a hybrid automaton for this thermostat can be given by Figure 2.1.

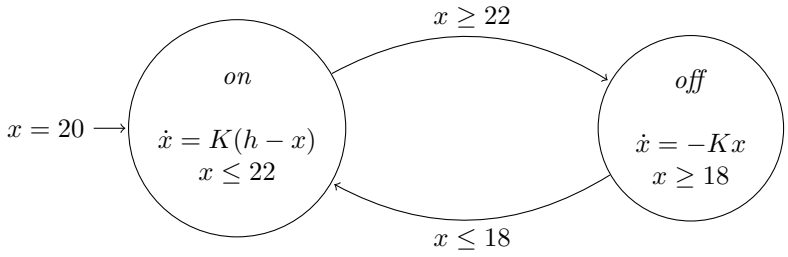


Figure 2.1: Hybrid automaton for Example 2.1.1.

2.1.1 Semantics of Hybrid Automata

The semantics of a hybrid automaton can be defined by an *operational semantics* that consists of one rule for discrete steps and one for continuous time steps.

Definition 2.1.2 (Discrete step semantics [Ábr15])
Let $A = (Loc, Var, Lab, Edge, Act, Inv, Init)$ be a hybrid automaton. The operational semantics for a discrete step in A is defined as:

$$(discrete) \quad \frac{(l,a,(v,v'),l') \in Edge \quad v' \in Inv(l')}{(l,v) \xrightarrow{a} (l',v')}$$

The above rule works as follows. If $(l,a,(v,v'),l') \in Edge$ and $v' \in Inv(l')$ this means that $(v,v') \in \mu$ (i.e. the guard of the transition is satisfied) and that the new valuation v' satisfies the invariant in the new location l'. Consequently, if these two conditions hold the automaton can be transferred from state (l,v) to (l',v') using the transition with label a.

Definition 2.1.3 (Continuous time step semantics [Ábr15])
Let $A = (Loc, Var, Lab, Edge, Act, Inv, Init)$ be a hybrid automaton. The operational semantics for a continuous time step in A is defined as:

$$(continuous) \quad \frac{f \in Act(l) \quad f(0) = v \quad f(t) = v' \quad t \geq 0 \quad f([0,t]) \subseteq Inv(l)}{(l,v) \xrightarrow{t} (l,v')}$$

The above rule states that if f is one of the activities of location l whose valuation at time point 0 is v and v' at the end of time step $t \geq 0$, and the valuations of the activity during that timespan satisfies the invariant of location l, we can transfer the automaton from (l,v) to (l,v'). Note that the automaton does not leave its current location, but only applies the activity to the current valuation.

With these two rules we can define the reachability problem for hybrid automata.

Definition 2.1.4 (Reachability [Ábr15])
Let $A = (Loc, Var, Lab, Edge, Act, Inv, Init)$ be a hybrid automaton. An execution step of A is either a discrete or a continuous step denoted by $\rightarrow = \xrightarrow{a} \cup \xrightarrow{t}$.
An execution π of A is a sequence of execution steps $\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \ldots$ where π_0 is an initial state. A state σ in A is reachable iff σ is an initial state or there exists an execution π in A with $\pi = \pi_0 \rightarrow \cdots \rightarrow \sigma$.

As stated earlier we are interested in finding out whether or not a hybrid system can reach a (possibly undesirable) state. Hence, the *reachability problem* is to find out if there exists an execution in a hybrid automaton that reaches a set of *bad states* $Bad = \{(b_1,v_1), \ldots, (b_n,v_n)\} \subseteq Loc \times \mathcal{V}$.

In general, the reachability problem for hybrid automata is *undecidable* due to the fact that hybrid automata are able to differentiate real points in time with infinite precision [HKPV98, Frä99]. However, *over-approximating* the reachable states of a hybrid automaton leads to a *semi-decidable* version of the problem. If the over-approximation does not reach the bad states we know that the precise set of reachable states does not as well and hence the automaton can be called safe. This will lead us to the notion of linear hybrid automata and flowpipe construction as an approach for a semi-decision procedure [LG09].

2.2 Linear Hybrid Automata

Linear hybrid automata differ from general hybrid automata in that the activities in linear hybrid automata are given by linear ordinary differential equations (linear

ODE) of the form

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

where $x \in \mathbb{R}^{|Var|}$ is a vector and A is a square $\mathbb{R}^{|Var| \times |Var|}$ matrix, the entries of which define coefficients for the continuous dynamics of each variable [LG09].

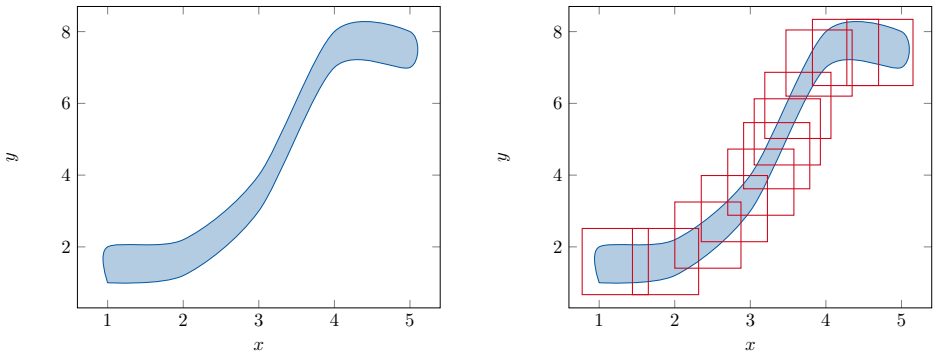
It can be shown that the solution to a linear ODE, i.e. the valuation in a location at time t can be given by computing

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

where x_0 is the initial value of x [LG09]. The term e^{tA} denotes the matrix exponential that can be compute as

$$e^X = \sum_{k=0}^{\infty} \frac{1}{k!} \cdot X^k \tag{2.3}$$

which converges for a square matrix X (or tA in our case) [Bel97] and can be approximated with moderate effort [ML03]. It can be deduced that for any set Y of valuations the reachable set at time t can be computed by computing $e^{tA} \cdot Y$ [LG09]. Consequently, we can *iteratively* build a sequence of reachable sets by repeatedly applying the matrix exponential to the solution of the last application and over-approximating the result, leading to a so called flowpipe. A first algorithm is given in Algorithm 2.2, a graphical version of the idea is given in Figure 2.2.



(a) Actual continuous dynamic. (b) Over-approximated continuous dynamic.

Figure 2.2: Flowpipe over-approximation.

The algorithm iteratively computes the next reachable states using `Reach(...)` in line 7 and adds the newly computed sets to the already computed reachable sets in line 6. If there are no new reachable sets the algorithm terminates. In the next section we will learn how `Reach(...)` is computed.

2.3 Flowpipe Construction

In the previous section we concluded that the set of reachable states of a hybrid automaton starting at an initial set of valuations X_0 can be computed by iteratively multiplying the current reachable set with a matrix exponential. However, we are yet to consider how to represent such a set of valuations. In this section we start

Algorithm 2.2 Simple Reachability Analysis [Ábr15]

```
1 Input: A hybrid automaton A with initial states Init_A
2 Output: Set Res of reachable states
3 R_new := Init_A;
4 Res := ∅;
5 while (R_new ≠ ∅){
6         Res := Res ∪ R_new;
7         R_new := Reach(R_new) \ Res;
8 }
9 return Res;
```

by representing sets of valuations by *convex polytopes* and define operations on this representation. Subsequently, we will see how we can over-approximate the reachable sets in such a way that it actually contains all reachable valuations in between two points in time. Finally, we will give an algorithm for reachability analysis that employs flowpipe construction and also considers discrete jumps in an automaton.

2.3.1 Representation

In this section we are going to use convex polytopes as a means of representing sets of valuations. Note that there are many other representations (e.g. boxes, ellipsoids, support functions etc.) that can be used coming with their own strengths and weaknesses w.r.t. *computational complexity* and *precision* [LG09].

Polytopes are commonly represented as either the convex hull of a finite set of vertices or as an intersection of a finite set of half-spaces. In this section we will choose the latter.

Definition 2.3.1 (Convex polytope [Zie12])
A half-space is defined as the set of points bounded by a hyperplane with normal n and an offset d:

$$h = \{x : x \cdot n \leq d\}$$

A polytope is the bounded intersection of finite number of half-spaces $\mathcal{H}$:

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

A common way of representing a polytope is as a matrix-vector combination

$$P = \{x : A \cdot x \leq b\}$$

where A is matrix constructed from normal vectors and b is a vector of offsets.

An example of a convex polytope is given in Figure 2.3.

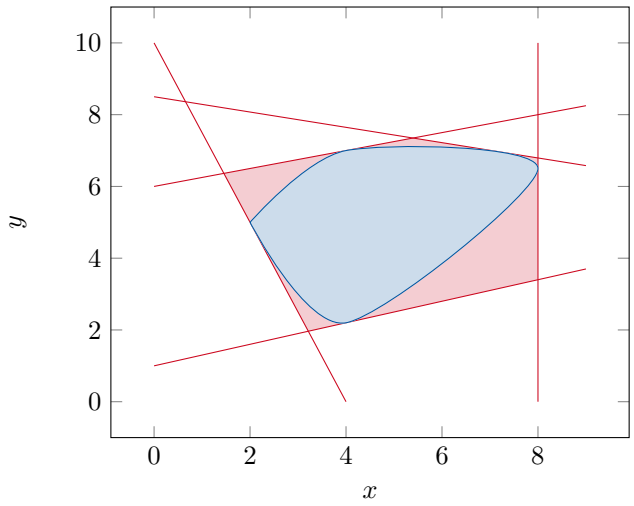


Figure 2.3: Two dimensional convex polytope (red) over-approximating another set (blue).

With the definition of a convex polytope we can define a number of useful operations that will aid us during flowpipe construction. Convex polytopes are closed under the following operations i.e. the results itself will be a convex polytope as well.

Intersection with a hyperplane [LG09]:
An *intersection* with a hyperplane will be used to obtain the set of values that fulfill a guard or invariant. The intersection with a hyperplane can be done by adding the corresponding half-space to the set of half-spaces.

Closure of the union with another polytope [LG09]:
The *union* will be used to unify the different flowpipe segments. The closure of the union of two polytopes combines their vertices in a single set and computes the *convex hull* of that set.

Linear transformation [LG09]:
Among other things, the *linear transformation* is used to apply the matrix exponential to the current set of valuations to obtain the reachable set of valuations for the next time step. The linear transformation of a polytope can be computed by either applying the transformation directly to the vertices of the polytope or, in case the transformation matrix is invertible, by applying it to the hyperplanes.

Minkowski sum [LG09]:
The *Minkowski sum* will be used to over-approximate the reachable valuations. The operation is performed with another geometric object, e.g. a box or sphere. The Minkowski sum is the set-theoretical equivalent to addition and given two set of position vectors A and B (e.g. defined by geometric objects) is defined as

$$A + B = \{x : x = a + b, \quad a \in A, b \in B\}.$$

Intuitively, one can imagine this as tracing the bounds of the original geometric object with the center of the e.g. box or sphere yielding a larger object. Note that this tracing is only correct if the object has its center in the origin. In hybrid reachability analysis this operation is often times referred to as *"bloating"*. An example is given in Figure 2.4.

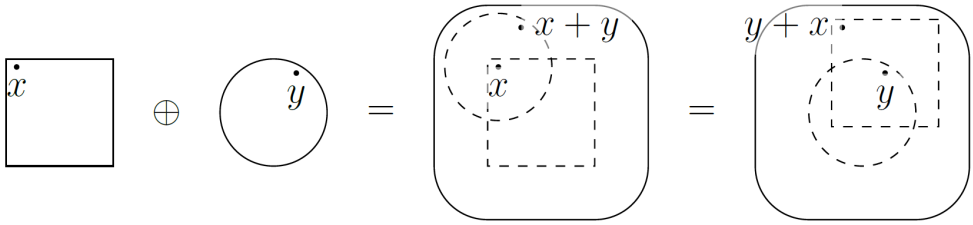
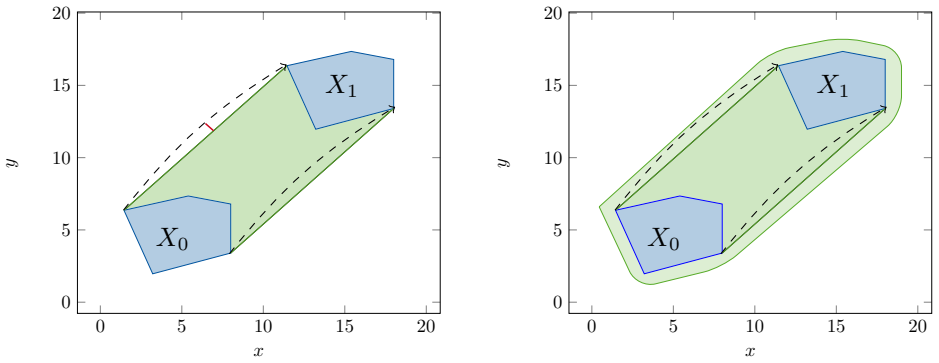


Figure 2.4: Minkowski sum of a square and a disk [LG09].

2.3.2 Over-approximation

As we stated earlier, determining the exact reachable set starting from a given set is undecidable. Nonetheless, over-approximating the reachable set is possible. Consider the set X_0 and its first time successor X_1 after time step t in Figure 2.5a where the dashed line denotes the actual continuous dynamics of the values. As we can see, some of these so called *trajectories* lie outside the closure of the union of X_0 and X_1 (green region & original polytopes) so we need over-approximation to include these trajectories in our set as they cannot be computed precisely.



(a) Reachable set without bloating.

(b) Reachable set with bloating.

Figure 2.5: Over-approximation of the initial set.

This over-approximation is done by applying the Minkowski sum with a bloating object to the convex hull of the union of X_0 and X_1, effectively including the missing trajectories (see Figure 2.5b). This leaves us with the question of how large such an object must be such that it is large enough to successfully over-approximate the precise set. There are multiple approaches for estimating the size of such an object. For example, one can use the *Hausdorff distance* to estimate the difference between

the actual trajectories and their linear approximations. The Hausdorff distance of two sets S and S' defines the maximum of the shortest paths from any point in S to any point in S' (red line in Figure 2.5a) [LG09]:

$$h(S,S') = max\{\underset{a\in S}{sup}\ \underset{b\in S'}{inf}||a-b||,\ \underset{b\in S'}{sup}\ \underset{a\in S}{inf}||a-b||\}$$

Note that this over-approximation is only done for the first segment of a flowpipe, because subsequent multiplication of this set with e^{tA} will maintain the over-approximation property for the rest of the computation [LG09]. Consequently, the *first segment* Ω_0 of a flowpipe for time interval $[0,t]$ starting with an initial state (l,p) consisting of a location l and variable valuation p is computed as

$$\Omega_0 = (conv(p \cup e^{tA} \cdot p) \oplus \mathcal{B}) \cap Inv(l)\ . \tag{2.4}$$

Here, A defines the activities of location l, $conv(\dots)$ computes the convex hull, $\mathcal{B}$ is a bloating object and $\oplus$ is the Minkowski sum operator [LG09].

All subsequent segments of the flowpipe in a location are given by the recurrence relation [LG09]:

$$\Omega_{i+1} = e^{tA} \cdot \Omega_i \cap Inv(l) \tag{2.5}$$

Note that the time step t is fixed at the beginning of flowpipe construction based on the problem instance. It cannot be altered during flowpipe construction without recomputing the bloating as changing the time step invalidates the bloating and therefore the over-approximation. As an example for such a flowpipe construction we consider the bouncing ball example, whose automaton and flowpipe can be found in Figure 2.6.

Example 2.3.1 (Bouncing Ball [Ábr15])
Imagine a ball with initial height $x > 0$ and velocity $v = 0$. Gravity with an acceleration of $\dot{v} = -9.81\ m/s^2$ immediately starts pulling downwards on the ball, accelerating it towards the ground. Upon reaching the ground the ball is going to bounce, i.e. the velocity of the ball is going to invert while also losing some of it due to the impact as the ball is elastic. While rising, gravity will again start pulling down on the ball decreasing its velocity until the ball reaches its apex and the ball will start falling back down.

In order to have a full flowpipe construction-based analysis algorithm we still have to consider what happens to the current flowpipe upon discrete *transitions*. In particular, we have to consider a transition's *guard* and its *effects*. Considering the bouncing ball automaton in Example 2.3.1 the transition is taken when the ball hits the ground, i.e. when its guard $x = 0 \wedge v \leq 0$ is satisfied, which in turn applies the effect to v and sets its new velocity to $v := -c \cdot v$.

A transition (l,a,μ,l') is enabled if there exists a set of valuations that satisfies the guard condition, i.e. the set is contained in μ. However, since valuations are represented by over-approximative geometric objects it may be the case that only some subset of an object is in μ and therefore actually satisfies the guard. For example in case of the bouncing ball the polytope that intersects the v axis meaning that $x = 0$ (i.e. the ball touches the ground) also contains some valuations of x that are larger than 0. These values do not intersect the guard and hence must be cut off. This is done by intersecting the half-spaces of the polytope with the half-spaces defined by the guard.

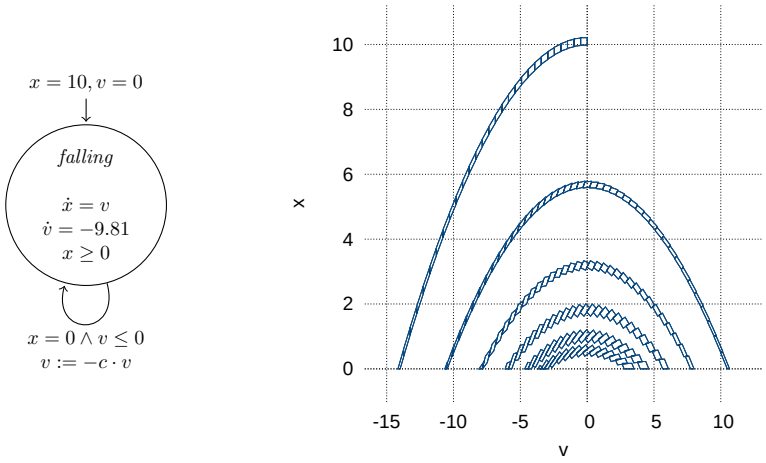


Figure 2.6: Bouncing ball automaton and flowpipe.

After we have computed the guard satisfying set the effect of the transition is applied to the set by inverting the valuation of v. In practice this is done by applying this reset as an affine transformation to the guard satisfying set.

The resulting set acts as a *new initial set* in the target location l' of the taken discrete transition. Consequently we have to compute a new initial set using Equation (2.4). This is due to the fact that in the new location the activities or so called flows may have changed. A changed flow may result in new trajectories that are outside our over-approximation and as a consequence we have to recompute a new bloating.

Now we have all tools needed to define a full flowpipe construction-based reachability analysis algorithm for hybrid automata (see Algorithm 2.3.2). First *Res* and R_{new} are initialized with the initial state set of automaton A in lines 3 and 4. *Res* will store all reachable sets computed during flowpipe construction whereas R_{new} will store sets that are used as starting points for future flowpipe constructions. The algorithm iteratively takes a `stateSet` from the set of states R_{new} in line 6 and constructs a flowpipe starting with that `stateSet` in line 7. When the flowpipe is constructed jump successors are computed in line 8. Note that the method `computeJumpSuccessor(...)` adds all newly computed reachable sets to R_{new} and *Res*. Computation terminates when either R_{new} becomes empty or a termination condition e.g. a maximum number of jumps is met.

Although this algorithm is suitable for linear hybrid automata and its subclasses the run time of this algorithm tends to explode with increasing number of variables/dimensions as the complexity of the operations on the state set (i.e. the polytope) increases. In fact, for some subclasses of linear hybrid automata the computation of reachable sets can be done much more efficiently than using flowpipe construction. In this thesis we aim for a dynamic approach that can decide what algorithm to use for reachability computation based on the automaton's current location. In the scope of this thesis we will employ this approach for timed automata which are presented in the next section.

Algorithm 2.3.2 Flowpipe construction-based reachability analysis

```
1  Input: A linear hybrid automaton A with initial states Init_A
2  Output: Set Res of reachable states
3  Res := computeInitialSegement(Init_A)
4  R_new := Res
5  while (R_new ≠ ∅ ∧ ¬termination_condition) {
6          stateSet := R_new.pop()
7          R' := computeFlowpipe(stateSet)
8          computeJumpSuccessor(R')
9  }
10 return Res
```

2.4 Timed Automata

Timed automata are a subclass of linear hybrid automata, whose purpose is to model real-time systems using real-valued variables called *clocks*. Clocks differ from variables in linear hybrid automata in that their activities are always 1, i.e. they progress at a *constant rate* [AD94]. Furthermore, a clock's valuation is always positive and can only be observed, i.e. compared to constants, or reset to zero after which it will continue to increase its value with constant rate 1. Consequently, clocks are used to measure the time spent in a certain location or set of locations as well as to regulate the order in time in which locations are visited by defining constraints over clocks.

Definition 2.4.1 (Clock constraints [Sri12])
Let $\mathcal{C}$ be a set of variables called clocks that range over $\mathbb{R}_{\geq 0}$. A clock constraint φ over $\mathcal{C}$ is defined by the grammar

$$\varphi := x \sim c \mid \varphi \wedge \varphi$$

where $x \in \mathcal{C}$, $c \in \mathbb{N}$ and $\sim \in \{<, \leq, =, \geq, >\}$. The set of all clock constraints is denoted as $\mathcal{CC}(\mathcal{C})$. A clock valuation is a function $v : \mathcal{C} \to \mathbb{R}_{\geq 0}$ that assigns values to clocks. We say that valuation v satisfies clock constraint φ, or $v \models \varphi$ for short, when replacing every occurrence of all $x \in \mathcal{C}$ in φ with its respective valuation $v(x)$ satisfies all constraints in φ.

Consequently, a timed automaton can be defined as a hybrid automaton whose variables are clocks that progress at constant rate 1 and whose location invariants and transition guards are *clock constraints*. The effect of a transition is a set of clocks that is going to be reset when taking the corresponding transition.

Definition 2.4.2 (Timed automaton [BK08])
A timed automaton A is a tuple $A = (Loc, \mathcal{C}, Lab, Edge, Act, Inv, Init)$ where

- *Loc is a finite set of locations.*
- *$\mathcal{C}$ is a finite set of clocks.*
- *Lab is a finite set of transition labels.*
- *$Edge \subseteq Loc \times \mathcal{CC}(\mathcal{C}) \times Lab \times 2^{\mathcal{C}} \times Loc$ is a finite set of edges, written as a tuple $(l,g,a,\mathcal{R},l')$ meaning that a transition labeled a from l to l' can be taken if guard $g \in \mathcal{CC}(\mathcal{C})$ is satisfied. Upon transitioning, all clocks $r \in \mathcal{R}$ are reset to 0.*

- *Act is a function assigning a set of time invariant activities to each location. Note than in this case this function is the constant function* 1*.*
- *Inv is a function that assigns a set of invariants* $Inv(l) \in \mathcal{CC}(\mathcal{C})$ *to each location. A timed automaton must leave its current location before the clock valuations invalidate the current location's invariant.*
- $Init \subseteq Loc \times \mathcal{I}$ *is a set of initial states where* $\mathcal{I} \in \mathcal{CC}(\mathcal{C})$ *is a set of clock constraints defining the initial clock valuations.*

Example 2.4.1 (Bolognese sauce)
More than for my achievements in hybrid automaton analysis I am known for my world class bolognese sauce. The secret of this sauce is to repeatedly reheat it which, according to popular belief, breaks down more amino acids and browns more sugar to enhance its savory, umami taste [MSJM14].

After mixing all the ingredients in the cooking pot the mixture is cooked for 30 minutes without stirring. After 30 minutes we enter a reheating cycle by starting to cool the sauce down. Cooling proceeds for 10 minutes during which every 4 minutes the sauce is stirred for 1 minute. After those ten minutes, we switch to reheating the sauce for 10 minutes stirring it every 4 minutes for 1 minute as well. We then switch back to cooling and the cycle starts over.

As always, a sauce is done when it's done.

A depiction of the corresponding timed automaton can be found in in Figure 2.7. Note that clocks that have to be reset are denoted by curly braces on the edge label.

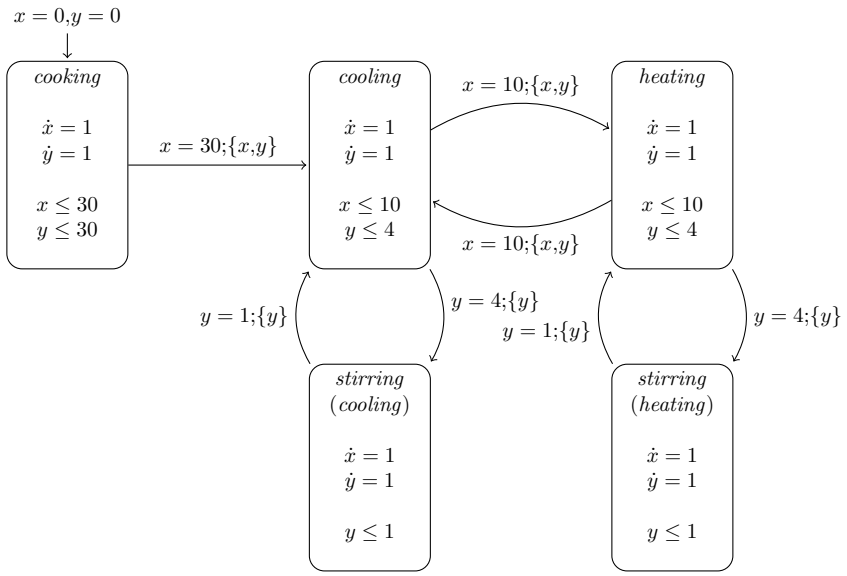


Figure 2.7: Timed automaton for Example 2.4.1.

2.4.1 Zone-based Analysis

Basically, there are two common approaches to computing the reachable set of states in a timed automaton. The first one divides the reachable state space into a finite set of

equisized regions in which the valuations cannot be distinguished by the automaton, thereby defining a finite region graph that is then analyzed. However, it can be shown that the number of regions, although finite, grows exponentially in the number of clocks [AD94].

Therefore, we are going to use the improved approach of *zone-based abstraction* to analyze a timed automaton's reachable state set [BY04]. Intuitively, a zone is a set of clock valuations that is defined by a *finite* set of clock constraints.

Definition 2.4.3 (Zones [BY04])
Let $\mathcal{C}$ be a set of clocks. A zone is a set of clock valuations defined by a finite conjunction of clock constraints of the form:

$$x \sim c$$
$$x - y \sim c$$

for x,$y \in \mathcal{C}$, $\sim \in \{\leq, <, =, >, \geq\}$ and $c \in \mathbb{Z}$

Figure 2.8 illustrates what a zone may look like.

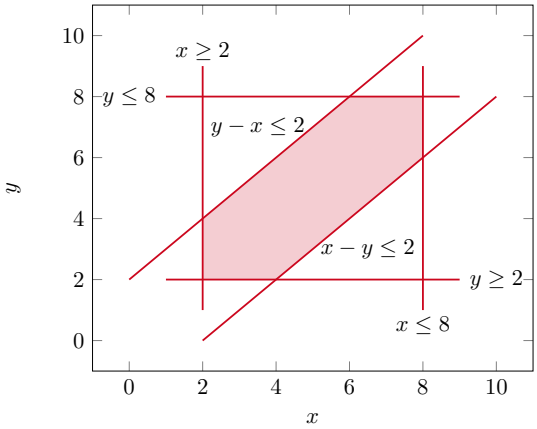


Figure 2.8: Example of a Zone.

Note that contrary to the definition of clock constraints in Definition 2.4.1 we allow the constant c to be an *integer* rather than a natural number and allow the constraints to be defined over *clock differences*. In Section 3.2 when we are going to talk about the implementation details of zones we will see that this will greatly simplify the data structure and operations needed to work with zones.

As we can see, zones are not much different from polytopes used for flowpipe construction-based reachability analysis. However, zones can exploit the fact that the clocks evolve at a fixed rate of 1 so that we neither have to bloat the zone, nor do we have to iteratively construct a flowpipe using costly matrix multiplications until we reach some time horizon or the location's invariant is invalidated. Instead, we can just *elapse* the entire time by moving the upper bounds of our zone ($x \leq 8$ and $y \leq 8$ in Figure 2.8) to the end of our time horizon while considering the location's invariants reducing the computation of reachable valuations in a location to a *single step*. Because the time horizon is merely given as a constant and invariants are given

as clock constraints of the form $x \sim c$ (see Definition 2.4.1) this computation comes down to altering the constants in the clock constraints defining our zone and thus can be done with very little effort.

Technical details on how to actually compute the elapsed zone will be subject of Section 3.2. An example of an elapsed zone can be found in Figure 2.9. Note that the amount of constraints needed to describe the original and the elapsed zone is the same.

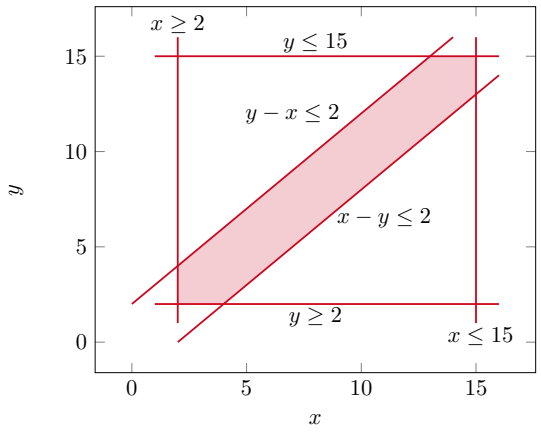


Figure 2.9: Example of zone in Figure 2.8 elapsed with time horizon 15.

Discrete jumps in zone-based reachability analysis are equally easy to compute since we can exploit the fixed rate of clock movement as well as the fact that the effect of a transition is a reset that sets clocks back to 0. Consequently, intersection with a guard can be done by adding the corresponding clock constraint to the constraints of our zone while the effect of a transition is a *projection* to the axes.

Accordingly, a full zone-based analysis algorithm looks very similar to a flowpipe construction-based analysis algorithm (see Algorithm 2.4.1). It is different in that it does not call `computeFlowipe(...)` in line 7, but calls the much more efficient method `computeElapsedZone(...)` that employs the above strategy.

Additionally, the implementation of `computeJumpSuccessors(...)` is altered to exploit the concept of zones.

Algorithm 2.4.1 Zone-based reachability analysis

```
1  Input: A linear hybrid automaton A with initial states Init_A
2  Output: Set Res of reachable states
3  Res := computeInitialSegement(Init_A)
4  R_new := Res
5  while (R_new ≠ ∅ ∧ ¬termination_condition) {
6          stateSet := R_new.pop()
7          R' := computeElapsedZone(stateSet)
8          computeJumpSuccessor(R')
9  }
10 return Res;
```

Chapter 3

HyPro

In the previous chapter we have seen that in order to perform analysis of hybrid systems we need an abstraction of the system, i.e. a hybrid automaton, as well as geometric representation of the current state in the form of a geometric object like for example a hyperplane polytope.

In this thesis we will make use of the HyPro [SÁMK17] toolbox that offers an implementation of a hybrid automaton as well as a large variety of different geometric objects that can be used to represent the current state of the automaton.

In Section 3.1 we will give a brief overview of HyPro's architecture considering the data structures and geometric representations it offers. Subsequently, in Section 3.2 we are going to augment HyPro's rich variety of geometric objects with a new representation called difference bound matrix that is suitable to efficiently represent zones as introduced in Definition 2.4.3.

3.1 Architecture

The HyPro toolbox is a free and open-source C++ library[1] that basically consists of two main components (see Figure 3.1).

The first big part are the *data structures* that contain implementations of a *Point*, *Half-space* and *Hybrid automaton*. The second big part are the *representations* that provide a large variety of *geometric objects* that all implement an interface of the same name to support interchangeability achieved by a converter that converts between the different representations.

Note that HyPro also offers its own reachability algorithm that we are not going to use in this thesis. Instead, we will use the HyDRA toolbox which offers a much more sophisticated analysis algorithm featuring multi-threading support among other useful features.

HyPro's implementation of a hybrid automaton is very close to the definition of a hybrid automaton given in Definition 2.1.1. A hybrid automaton in HyPro consists of a list of locations (with flows and invariants) and transitions (with guards and resets) as well as a list of initial and bad states.

However, HyPro's hybrid automaton components have been altered to support *multi-sets*. This means that we are able to store more than one flow function and in-

[1] A current version of the library can be found at https://github.com/hypro/hypro

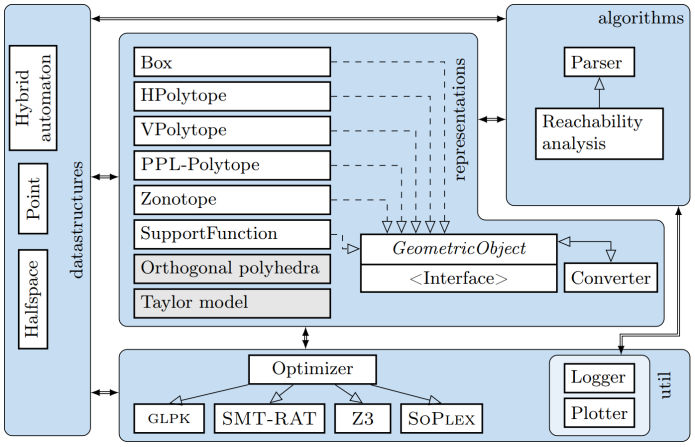


Figure 3.1: Basic architecture of the HyPro library [SÁMK17].

variant per location as well as more than one guard and reset per transition. Similarly, an automaton's state can store more than one state set representation to describe its variable valuations. This effectively enables us to separate the automaton's variable set into smaller subsets that can then be analyzed independently as we will see in Chapter 4.

All of HyPro's state set representation implement the `GeometricObject` interface which ensures that all representations implement the same basic functions that allow them to be used in hybrid system reachability analysis (see Figure 3.2).

Looking back at Section 2.3.1 the bare minimum of operations necessary are *intersection* with a hyperplane, *convex union* with another geometric object, *linear transformation* and *Minkowski sum*. Nonetheless, the `GeometricObject` interface requires state set representations to provide a few more functions to either allow for convenience features like plotting state sets or performance enhanced computations, e.g. intersection with multiple hyperplanes.

The representations differ from each other in how *precise* they describe a set of values and how *complex* the basic operations intersection, convex union, linear transformation and Minkowski sum are to compute. For example, a box merely can be represented as a minimal and maximal point which makes operations like Minkowski sum very fast and easy to compute. However, boxes are very imprecise when used to describe arbitrary sets.

Hyperplane polytopes (HPolytopes) on the other hand can describe a set of values more precise the more hyperplanes it uses, but apart from the operations intersection and linear transformation, all other operations are hard to compute.

A trade off between precision and complexity can be achieved by support functions. In theory, support functions describe a set of values exactly as it can be seen as an HPolytope with an uncountable number of hyperplanes. Naturally, it is impossible to store an uncountable number of constraints so hyperplanes are computed on-the-fly where needed so additional precision directly correlates to increased complexity. Operations on support functions are stored as a sequence of operations on the underlying hyperplanes that are not applied directly. Actual computational overhead arises when we have to check properties of the support function, e.g. whether or not a point

is contained in the support function.

<<interface>>
GeometricObject

```
dimension( ) : int
empty( ) : bool
vertices( ) : vector<Point>
satisfiesHalfspace(Halfspace) : pair<bool,GeometricObject>
satisfiesHalfspaces(matrix,vector) : pair<bool,GeometricObject>
project(vector<int>) : GeometricObject
linearTransformation(matrix) : GeometricObject
affineTransformation(matrix,vector) : GeometricObject
minkowskiSum(GeometricObject) : GeometricObject
intersectHalfspace(Halfspace) : GeometricObject
intersectHalfspace(matrix,vector) : GeometricObject
contains(Point) : bool
unite(GeometricObject) : GeometricObject
```

Figure 3.2: `GeometricObject` interface.

3.2 Difference Bound Matrices

One of the main contributions of this thesis is an implementation of a state set representation for *zones* (see Definition 2.4.3) called *difference bound matrix* [BY04] that satisfies the `GeometricObject` interface of HyPro (see Figure 3.2). The representation exploits the fact that clocks progress at constant rate 1 which restricts the maximum number of constraints needed to represent an arbitrary zone to be *quadratic* in the number of clocks [BY04]. This will enable us to represent clock valuations exactly without the need of over-approximation while requiring *less memory* and offering *optimized operations* during reachability computation.

Recall that a zone is a finite conjunction of clock constraints of the form $x \sim c$ and $x - y \sim c$ where x,$y \in \mathcal{C}$ are clocks, $\sim \in \{\leq , < , = , > , \geq\}$ and $c \in \mathbb{Z}$. The most efficient form to store these constraints is in form of a special matrix that contains bounds on the difference between values of two clocks, hence the name difference bound matrix (DBM) [Bel57, Dil90]. The basic idea is that row and column index of a DBM specify the clocks compared while the value at that position is the difference between the clock values.

However, the above definition contains constraints of the form $x \sim c$ which technically is not a difference between two clocks and therefore it is unclear where to place the bound of this constraint in the difference bound matrix. It is for that reason that we introduce a zero clock $\mathbf{0}$ with constant value 0 so that we can represent clock constraint uniformly as $x - y \sim c$ where x,$y \in \mathcal{C}_\mathbf{0} := \mathcal{C} \cup \{\mathbf{0}\}$. This has the convenient effect that we consequently can reduce the number of comparison operators needed to represent a zone. For example equality constraints of the form $x = c$ can now be represented as the conjunction of $x - \mathbf{0} \leq c$ and $\mathbf{0} - x \leq -c$. This leads us to the following lemma:

Lemma 3.2.1 (Uniform zone representation [BY04])
Let Z be a zone according to Definition 2.4.3, then Z can be rewritten as a conjunction of constraints of the form

$$x - y \preceq c$$

for x,$y \in \mathcal{C_0}$, $\preceq \in \{<, \leq\}$ and $c \in \mathbb{Z}$. The maximum number of constraints needed to describe zone Z is $|\mathcal{C_0}|^2$.

Consequently, we can store clock constraints in a $|\mathcal{C_0}| \times |\mathcal{C_0}|$ matrix described by the following definition:

Definition 3.2.1 (Difference bound matrix [BY04])
Let Z be a zone whose constraint have been rewritten to fit Lemma 3.2.1 over clocks $C_\mathbf{0} := \{x_0 := \mathbf{0}, x_1, x_2, ..., x_n\}$, then Z can be represented as a $|\mathcal{C_0}| \times |\mathcal{C_0}|$ difference bound matrix D whose elements (denoted as D_{ij} for index i,j) are constructed by the following rules:

- *For each difference $x_i - x_j \preceq n$ of Z, let $D_{ij} = (n, \preceq)$.*
- *For each unbounded difference $x_i - x_j$ let $D_{ij} = \infty$*

An example DBM for the zone in Figure 2.8 can be found in Figure 3.3.

$$\begin{array}{c} \\ \mathbf{0} \\ x \\ y \end{array} \begin{array}{c} \begin{array}{ccc} \mathbf{0} & x & y \end{array} \\ \begin{pmatrix} (0, \leq) & (-2, \leq) & (-2, \leq) \\ (8, \leq) & (0, \leq) & (2, \leq) \\ (8, \leq) & (2, \leq) & (0, \leq) \end{pmatrix} \end{array}$$

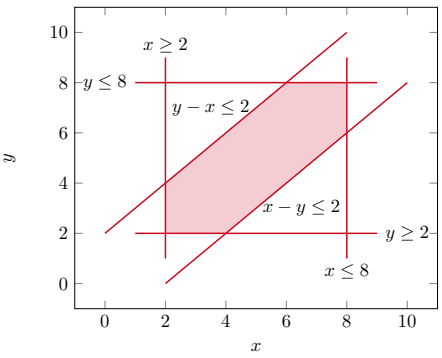


Figure 3.3: Example difference bound matrix.

Note that diagonal entries in a DBM are always $(0, \leq)$ because a clock can not run faster/slower as itself, i.e. $x_i - x_i \leq 0$. As we want to run algorithms on a DBM, but the actual DBM entries are a non standard construct rather than numbers we define basic operations on DBM entries.

Definition 3.2.2 (DBM entry operations [BY04])
Let D be a DBM according to Definition 3.2.1 and let m,$n \in \mathbb{Z} \setminus \{\infty\}$ as well as $\preceq \in \{<, \leq\}$. We define the following operations on DBM entries:

- *Comparison:*

  1. $(n, \preceq) < \infty$

2. $(m, \preceq) < (n, \preceq)$ *if* $m < n$
3. $(n, <) < (n, \leq)$

- *Addition:*

1. $(n, \preceq) + \infty = \infty$
2. $(m, \leq) + (n, \leq) = (m + n, \leq)$
3. $(m, <) + (n, \preceq) = (m + n, <)$

A DBM is called *canonical* [BY04] if none of its entries can be changed without changing the zone it defines. Intuitively, one can imagine that all constraints in a canonical DBM are *tight* in the sense that their graphical representation touches the zone. Note that the DBM defined in Figure 3.3 is canonical. This property is very important because it simplifies operation like intersection and emptiness check and our goal is to maintain the canonicity property throughout the entire computation.

In the following section we are going to investigate the implementation of various operations on DBMs.

3.3 Operations on Difference Bound Matrices

To use DBMs in HyDRA's reachability analysis algorithm it must be a HyPro representation satisfying the `GeometricObject` interface. In this section we are going to explore DBM specific operations such as elapsing and constraint intersection.

The attentive reader will notice that some of the important operations of the `GeometricObject` interface e.g. linear transformation are missing in this section. This is due to the fact that these operations are not directly applicable to DBMs as they were not designed to support these operations. They are replaced by DBM specific methods that are suitable for the intended use of e.g. linear transformation. Recall that linear transformation is intended to compute time successors as well as transition resets. As clocks move at a fixed rate of 1 and can only be reset to 0, the linear transformation is replaced by the new DBM specific methods elapse, shift and reset.

For the sake of interface completeness the methods not directly applicable are nonetheless provided by first converting the DBM to a representation that supports the operation, applying the operation and then converting back to DBM. Note that this may introduce over-approximation.

We already learned that DBMs can be used to represent zones and that zones are a means to capture clock valuations of timed automata. Consequently, there are a few timed automaton specific operations that are not part of the `GeometricObject` interface (e.g. elapsing) because they are not applicable to general state set representations.

Here, we are going to introduce the operations *intersection with clock constraints* as well as *elapsing*, *shifting* and *resetting* of clocks.

3.3.1 Intersection with Clock Constraints [BY04]

Intersecting zones with clock constraints models the intersection with transition guards and location invariants during reachability analysis.

Contrary to other HyPro representations where intersections can be performed with arbitrary hyperplanes zones can only be intersected by clock constraints (see Definition 2.4.1) because that is the only form of constraint the underlying timed automaton allows. Consequently we can exploit the general form of the constraints to efficiently manipulate the DBM that describes our zone (see Algorithm 3.3.1).

Algorithm 3.3.1 `intersectConstraint`(D,$x_i - x_j \preceq c$)

```
1  Input: DBM D, clock constraint x_i − x_j ⪯ c
2  Output: DBM D intersected with clock constraint x_i − x_j ⪯ c
3  if (D_ji + (c, ⪯) < 0){
4    D_00 = (−1, ≤)
5  }
6  else if ((c, ⪯) < D_ij){
7    D_ij = (c, ⪯)
8    // restore canonicity
9    for (k = 0...n){
10     for (l = 0...n){
11       if (D_ki + D_il < D_kl){
12         D_kl = D_ki + D_il
13       }
14       if (D_kj + D_jl < D_kl){
15         D_kl = D_kj + D_jl
16       }
17     }
18   }
19 }
```

Recalling the definitions of clock constraints and DBMs we trivially know which entry of the DBM has to be altered. Given a clock constraint of the form $x_i - x_j \preceq c$ and a DBM D we check if $(c, \preceq) < D_{ij}$ using Definition 3.2.2. If this inequality does not hold the constraint does not further constrain our zone and we are already done. Otherwise, we replace entry D_{ij} with $(c, \preceq)$ (see line 7).

The attentive reader may notice that performing such an operation may invalidate the *canonicity* property of a DBM (see Figure 3.4) or make the zone empty altogether.

Therefore, after intersection with the new constraint we explicitly have to check and if necessary restore the canonicity property of the DBM (see lines 9 to 18).

Fortunately, the problem of possible *emptiness* is easier to handle. Before we actually intersect the DBM with the new constraint we first check whether the intersection would move the upper (lower) bound of the affected clocks lower (higher) as their lower (upper) bound because in that case we can deduce that the zone becomes empty after intersection. If the DBM becomes empty we neither perform the intersection nor do we have to consider maintaining canonicity. Instead, we abuse the otherwise unused entry D_{00} by replacing it with $(-1, \leq)$ as an indicator that the underlying zone is empty (see line 4). Consequently, we will save a lot of time when performing a future emptiness check by first checking whether the first entry of the DBM is $(-1, \leq)$ and in that case skip the emptiness computation since we already know the zone is empty.

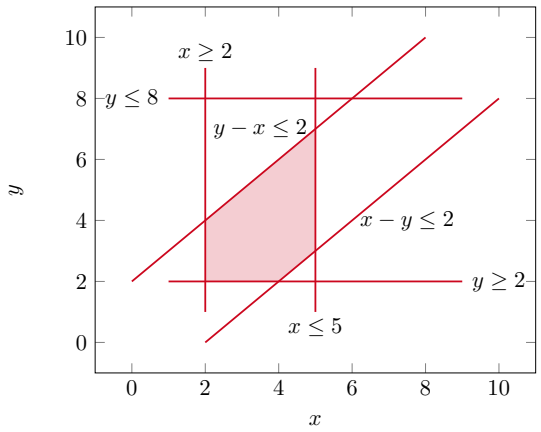


Figure 3.4: Non-canonical DBM from Figure 2.8 after intersection with $x - \mathbf{0} \leq 5$ ($y \leq 8$ is not tight).

3.3.2 Elapse [BY04]

Elapsing of zones models the passage of time in a location during which clocks will tick at rate 1, i.e. the reachable zone of clock valuations by staying in a location (see Algorithm 3.3.2).

Algorithm 3.3.2 `elapse`(D)

```
1 Input: DBM D
2 Output: Elapsed DBM D
3 for (i = 0...n){
4   D_i0 = ∞
5 }
```

Starting from a DBM D representing the initial zone this operation is computed by removing the upper bounds of all clocks by setting D_{i0} to ∞ (see line 4).

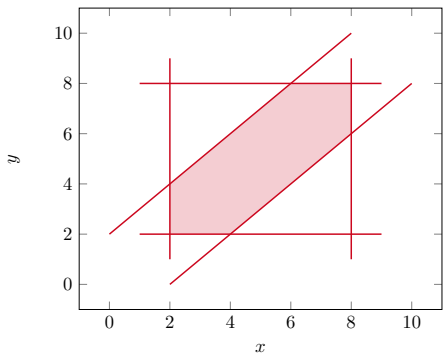
Note that elapsing a zone maintains canonicity. As all upper bounds are moved to ∞ and the DBM was canonical before the operation this means that none of the upper bounds in the DBM can be changed after the operation without changing the zone it defines, which is the definition of canonicity we introduced earlier.

3.3.3 Shift [BY04]

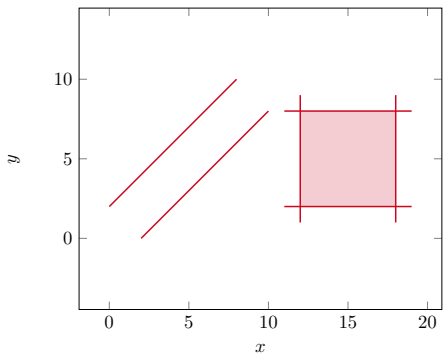
Shifting of a clock is the act of adding or subtracting a clock with a value, effectively moving the zone a certain distance in the direction of the specified clock. In this thesis we are going to use a slightly altered version of the shifting operation found in [BY04], but note that an implementation of the original shift can also be found in HyPro.

The original shift operation only shifts one clock by a given distance and therefore moves it along one of the clock axes. Consequently, shifting a zone along one of the

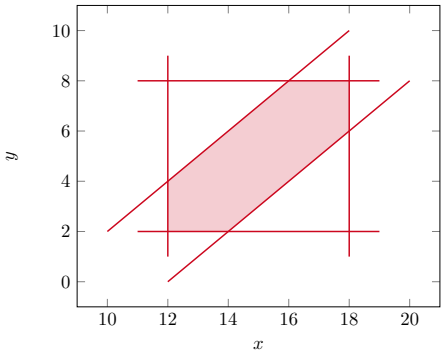
clock axes changes the difference between clocks and therefore raises the need to also move the diagonals in that direction (see Figure 3.5).



(a) Original DBM



(b) Shift of upper and lower bound of x by 10



(c) Shift of diagonals.

Figure 3.5: Example of a shift operation of x by 10 as described in [BY04].

In this thesis we implement a *uniform shift* that adds or subtracts all clocks with a value and hence the difference between clocks stays the same. This greatly simplifies the shift operation because we only have to alter the DBM entries in the first row and column to account for the lower and upper bounds of the clocks and make sure that all clock values stay positive (see Algorithm 3.3.3 lines 5 and 7). Recall that lower bounds in a DBM are represented by negative numbers so we have to subtract the distance for those bounds (see line 6).

We will use the uniform shift to model *clock ticks* i.e. the valuation of clocks after one time step and consequently this operation enables us to use DBMs in flowpipe construction-based reachability analysis that always considers the valuation of variables at one specific time step.

Algorithm 3.3.3 `uniformShift`(D,t)

```
Input: DBM D, distance t
Output: DBM D with all clocks shifted by t
for (i = 0...n){
   D_i0 = D_i0 + (t, ≤)
   D_i0 = max(D_i0,(0, ≤))
   D_0i = D_0i + (−t, ≤)
   D_0i = min(D_0i,(0, ≤))
}
```

3.3.4 Reset [BY04]

The effect of a transition in timed automata is the reset of some subset of clocks to 0. Intuitively, this operation is the projection of the zone to the axes of the clocks that are not reset (see Figure 3.6).

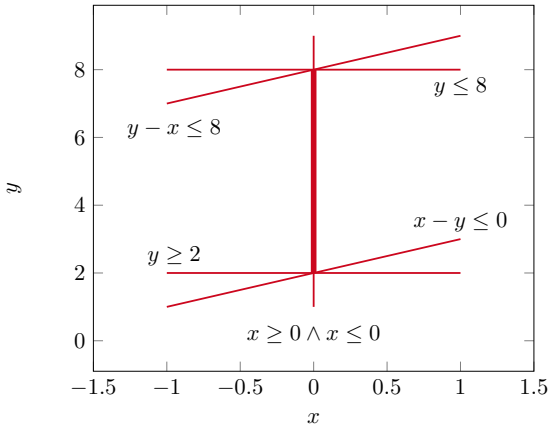


Figure 3.6: Zone from Figure 2.8 after reset of x to 0.

Suppose we want to reset clock x in DBM D to 0, then the operation begins with first setting the upper and lower bound of x to 0 by setting $D_{x0} = D_{0x} = (0, \leq) = D_{00}$ followed by moving the diagonal constraints that are defined over x to the upper/lower bound of the other variable in that constraint, i.e. $D_{xi} = D_{0i}$ and $D_{ix} = D_{i0}$. Pseudo code of the reset operation can be found in Algorithm 3.3.4.

Algorithm 3.3.4 `Reset`(D,x)

```
Input: DBM D, clock x to be reset
Output: DBM D with clock x reset to 0
for (i = 0...n){
   D_xi = D_0i
   D_ix = D_i0
}
```

Chapter 4

HyDRA

In the previous chapters we have discussed flowpipe construction-based analysis and the HyPro toolbox. In this chapter we are going to improve on HyDRA's flowpipe construction-based analysis algorithm that uses the tools offered by HyPro. The overarching goal of this part of the thesis is to construct a framework that supports *variable set separation*, i.e. independent analysis of *syntactically independent subsets* of the automaton's variables to reduce computational complexity.

The ability to perform independent analysis of the subspaces defined by the variable subsets enables us to take a more sophisticated approach based on the *context* the subspaces define, e.g. use the previously introduced difference bound matrix if the variable subset solely consists of clocks.

In Section 4.1 we are going to give a brief overview of HyDRA's architecture and its flowpipe construction-based reachability algorithm. Thereafter, we are going to introduce a decision entity that is able to determine a variable set separation of syntactically independent subspaces. Furthermore, the decision entity is able to analyze the subspaces and recommend a preferred representation to represent that subspace. Subsequently in Section 4.3 we are going to introduce a context-based worker that is able to dynamically choose an algorithm used for flowpipe construction in a location from a set of algorithms called contexts. Consequently in Section 4.4 we present the default context, a context that performs the general flowpipe construction algorithm presented in Algorithm 2.3.2. At the end of this chapter we present a timed context that is able to perform optimized reachability analysis in a timed automaton setting.

4.1 Architecture

HyDRA is an extensive implementation of the flowpipe construction algorithm presented in Algorithm 2.3.2 that has been extended by various perks and features. These features for example include multi-threading support using a thread pool pattern [Neu16, SÁ18b] or counter example guided refinement using backtracking [Hüt16, SÁ18a]. An overview of HyDRA's basic structure can be found in Figure 4.1.

At the heart of HyDRA sits the *reachability procedure* (yellow). In its core it maintains a queue of so called *tasks* that contain the current state (location and variable valuation) of the automaton alongside parameters for analysis that are processed by *workers* in a multi-threaded fashion.

The procedure starts by creating a hybrid automaton based on a specification file by using HyPro's integrated parser for automaton model files (gray). It continues by creating initial tasks based on the initial states of the created automaton and enqueues them in the task queue (red). Subsequently, the workers are created and added to the *worker pool* (blue) after which the reachability procedure will wait for the workers to finish processing the tasks. When all tasks are processed, the workers are terminated and the result is plotted.

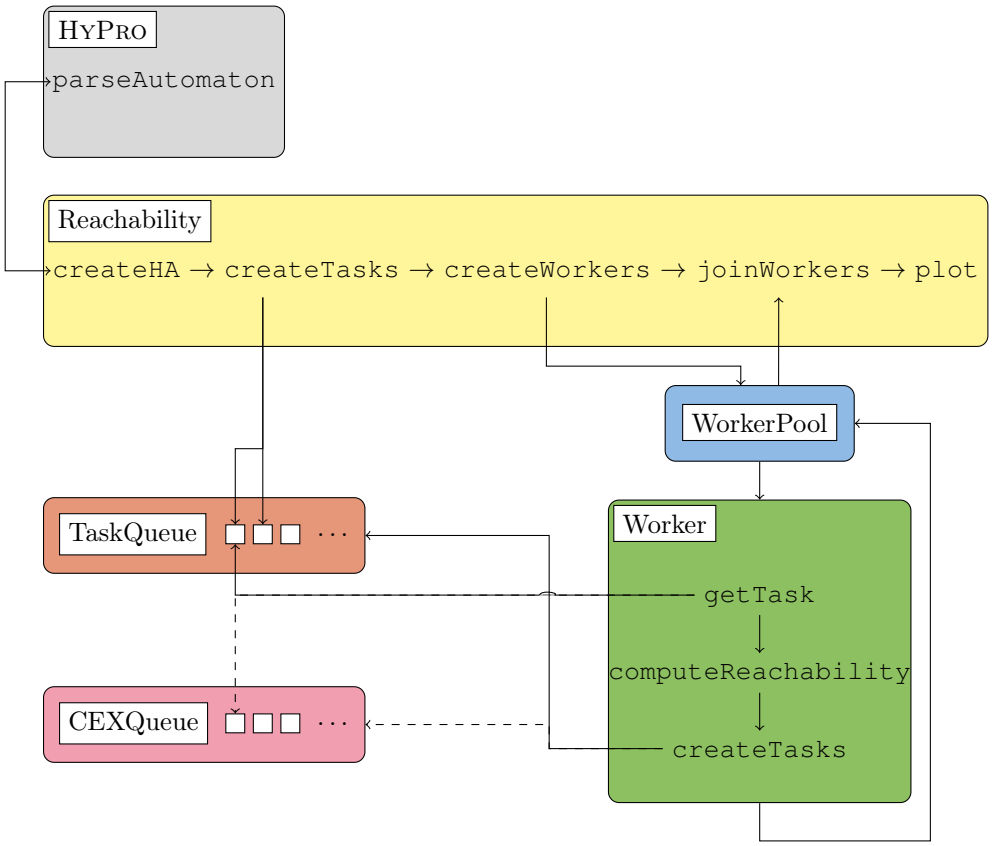


Figure 4.1: HyDRA architecture.

A *worker* (green) begins his work by popping the first available task from the task or counter example queue (magenta). Based on the automaton's state in that task a worker computes the reachable flowpipe for location and starting valuation defined by the state. When a worker is done computing the flowpipe in a location discrete jump successors are computed, wrapped in a task and enqueued in the task queue after which the worker goes back into the pool to wait for the next available task.

If a bad state was hit during flowpipe construction a worker wraps the hitting state in a task and enqueues it in the counter example queue. The next worker available can then use the counter example task to first backtrack and then recompute the flowpipe with finer analysis parameters, e.g. a more precise representation or smaller time step to check whether or not the counter example was spurious.

In the current version of HyDRA there is only one worker type. This worker performs general flowpipe analysis and the worker's implementation contains the whole reachability algorithm for a location which currently hinders rapid prototyping and extensibility.

In case someone wants to implement an algorithm to analyze timed automata one has to copy the whole worker implementation to a new worker type and change the code where needed to fit timed automata. Recalling the flowpipe algorithms for linear hybrid automata (Algorithm 2.3.2) and timed automata (Algorithm 2.4.1) we can see that they only differ in the way they compute continuous successors so copying the whole worker implementation for very few changes is extremely inefficient and hard to maintain. Changes to how tasks are created or how the underlying reachability tree for backtracking computation is handled would have to be changed in every single worker.

In Section 4.3 we will construct a new type of worker that is way more extensible and supports rapid prototyping.

4.2 Decision Entity

As stated earlier we strive for an implementation of HyDRA that supports variable set separation to perform less complex reachability analysis in *independent subspaces*, but we are yet to define how a hybrid automaton's variables can be decomposed such that a reachability algorithm still produces valid results.

This section will start by formally describing what variable set separation in a hybrid automaton is, how it can be achieved and how an algorithm has to proceed in order to produce valid results. We introduce the idea of a *decision entity* that is able to compute a variable *decomposition* and furthermore *classify* the subspace defined by a variable subset, e.g. whether it solely consists of clock variables. This classification enables us to *dynamically* decide on a representation for a subspace that may be better suited for analysis, e.g. difference bound matrices for timed subspaces.

4.2.1 Variable Set Separation and Automaton Decomposition

In flowpipe construction-based analysis complexity increases in the number of variables and even automata modeling smaller programs ($\backsim$20 variables) can be a significant challenge. The reason for this is that the more variables are involved in modeling complex behavior, the higher the dimensionality of the underlying state set representation becomes. Since the complexity of operations on state set representations is directly correlated to the number of dimensions, more variables entail longer run times and higher memory consumption.

To reduce the effects of high dimensionality recent efforts proposed the idea of *variable set separation* [SNÁ17, CS16]. The approach is based on the assumption that in most scenarios the evolution of a variable's valuation, besides the common notion of time, only depends on a subset of the other variables in the scenario, if any. Consequently, as long as variables do not influence each other we are able to analyze them separately from one another while the iterative nature of flowpipe computation considers the common notion of time passing synchronously in the respective subspaces.

This leads us to the concept of *syntactical independence* of variable sets. Intuitively, this syntactical independence states that two variables are syntactically in-

dependent if they do not occur in the same predicate (function resp.) or are not dependent to one another via transitivity in the predicates (functions resp.).

Let X be a set of variables in a hybrid automaton, $Pred_X$ the set of predicates i.e. guards and invariants in that automaton and $X_1 \cup \cdots \cup X_n$ a decomposition of X into disjoint subsets. The variable subsets are called syntactically independent if all predicates $\varphi \in Pred_X$ (similarly for jump resets and flows) are decomposable to a conjunction $\varphi = \varphi_1 \wedge \cdots \wedge \varphi_n$ where each φ_i is defined over the respective variable subset X_i [SNÁ17]. Note that the decomposition must be the same for all $\varphi \in Pred_X$, i.e. it is not allowed that e.g. guards are decomposed over a different variable decomposition than invariants.

If for an automaton such a decomposition into syntactical independent variable sets exists, we are able to represent the automaton's state $(l,p) \in Loc \times \mathcal{V}$ by its *projective representation* $(l, p_1, \ldots, p_n)$ where each p_i is the projection of state set p to the variables in variable subset X_i. In general however, projecting p to subspaces p_i introduces additional *over-approximation* (see Figure 4.2), because the implicit connection between the subspaces is lost, i.e. the combination of the subspaces by computing the *cartesian product* $p_1 \times \cdots \times p_n$ does not yield the original state set p [SNÁ17].

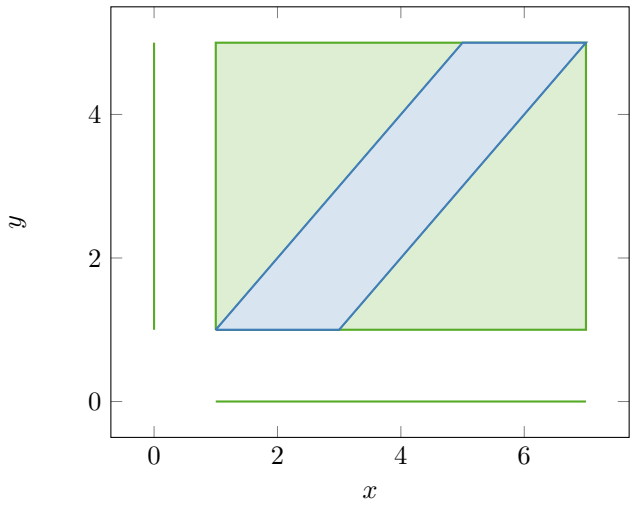


Figure 4.2: Original polytope p (blue) versus cartesian product of projections (green).

A way to account for this over-approximation is to either use boxes, as the cartesian product of the projections of a box yields the box itself, or use a smaller time step to limit the absolute over-approximation error introduced.

Recall the definitions of flowpipe construction-based analysis introduced in Section 2.3.2. In a flowpipe construction-based reachability algorithm based on variable set separation we extend these definitions to work on the projective representation of a state.

Definition 4.2.1 (Automaton decomposition [SNÁ17])
Let H be a hybrid automaton with variables X. Further let $X_1, \ldots, X_n$ be a decomposition of X into syntactically independent variable sets.

Any flow function in H of the form $\dot{x} = Ax$ can be decomposed into

$$\bigwedge_{i=1}^{n} \dot{X}_i = A_i X_i,$$

and any reset function of the form $x' = Ax$ can be decomposed into

$$\bigwedge_{i=1}^{n} X_i' = A_i X_i$$

where each of the A_i is a block matrix that defines the flow (reset resp.) for each of the syntactically independent variable subsets.

Similarly, any invariants or guards $\varphi \in Pred_X$ can be decomposed into

$$\bigwedge_{i=1}^{n} \varphi_i, \qquad \varphi_i \in Pred_{X_i}.$$

Consequently, flowpipe construction for subspaces can be defined as follows.

Definition 4.2.2 (Subspace flowpipe [SNÁ17])
Let H be a hybrid automaton with variables X. Let $X_1, \ldots, X_n$ be a decomposition of X into syntactically independent variable sets, $(l, p_1, \ldots, p_n)$ the projective representation of a state in H and all functions and predicates decomposed as described in Definition 4.2.1. Furthermore, let $\mathcal{B}_i$ be the projection of a bloating object to variable set X_i.

Then the first segment of a flowpipe for the subspace given by variable set X_i can be defined as

$$\Omega_{0,i} = (conv(p_i \cup e^{tA_i} \cdot p_i) \oplus \mathcal{B}_i) \cap Inv(l)_i.$$

Accordingly, any subsequent segments of the flowpipe for the subspace defined by variable set X_i are given by the recurrence relation

$$\Omega_{j+1,i} = e^{tA_i} \cdot \Omega_{j,i} \cap Inv(l)_i.$$

For the computation of jump successors, we require that a transition can only be taken when the respective guard is satisfied in *all* subspaces. In a similar vein an invariant is no longer satisfied, if *one of* the subspaces violates its respective invariant.

The only puzzle piece missing at this point is an automated way to obtain a syntactically independent variable set separation. We propose a simple undirected graph based algorithm that computes a separation of variables into a *maximal decomposition*, i.e. there is no decomposition that has more disjoint subsets and respects syntactical independence as described above. Keep in mind that this is only one approach for automated variable decomposition and that it is unclear whether it is the best approach. A quick rundown of possible other decompositions will be part of Chapter 6.

The algorithm works as follows (for pseudo code see Algorithm 4.2.1). The algorithm creates an empty graph and then adds a node for each variable in the automaton (line 7). Thereafter, the algorithm parses the automaton to find the dependencies between variables by investigating each function (e.g. line 12) and constraint (e.g. line 17) and if two variables occur in the same function (constraint resp.) an edge between the corresponding nodes is added in the graph (e.g. line 19) to imply that there is a dependency between these two variables. When the whole automaton is

parsed the connected components of the graph (line 48) are computed as a set of set of indices where each subset describes one of the connected components. Two variables must be in the same subspace if they are in the same connected component because a connection in the graph implies dependency.

4.2.2 Decomposition Classification

When an automatons variable set is decomposed into syntactically independent subspaces it opens up manifold possibilities for *optimization*. One of these optimizations is to *classify* a subspace to apply specialized representations and algorithms to that subspace.

Assume we have a hybrid automaton with variables x and y that depend on each other and clocks c_1 and c_2 that are neither dependent to each other, nor to x and y. Consequently, Algorithm 4.2.1 would suggest a decomposition of $X = \{x,y,c_1,c_2\}$ into $X_1 = \{x,y\}$, $X_2 = \{c_1\}$ and $X_3 = \{c_2\}$ for individual subspace computations.

The attentive reader will notice that X_2 and X_3 solely consist of clock variables, so it is a wise choice to not use the representation that would be used without decomposition for variable set X, e.g. a support function, but instead switch to a difference bound matrix to analyze these two subspaces to save both time and memory.

We introduce three possible classifications for a *subspace type*.

Linear A *linear classification* for a subspace states that the variables have no exploitable behavior to our knowledge, i.e. they behave like normal flowpipe variables whose flow is described by linear ODE and are analyzed as such.

Timed A *timed classification* for a subspace states that the variables of the subspace are clocks and hence can be analyzed using difference bound matrices that are both faster and more precise than general representations. A subspace is called timed if all the variables in it evolve at constant rate 1, invariants and guards of outgoing transitions are clock constraints and resets, if present, are resets to 0.

Note that a location is called *timed* if all subspaces are timed. Furthermore, an automaton is called timed, if all locations are timed. A timed location's flowpipe can be computed in one step by *zone elapsing* (see Section 2.4.1).

Discrete A *discrete classification* for a subspace states that the variables of the subspace have no continuous evolution, i.e. the flow of all variables is 0 and hence flowpipe construction can be skipped. Furthermore, since the variable valuations do not evolve over time intersection with invariants and guards has to be computed only once when entering a location as the result of that intersection never changes.

With subspace classifications and an algorithm for variable set separation Figure 4.3 presents a singleton class that offers this functionality in the HyDRA toolbox.

4.3 Context-Based Worker

In the previous section we introduced a decision entity that is able to compute and classify a subspace decomposition. To tap the full potential of the decision entity we implement a new context-based approach in a *context-based* worker that will replace

Algorithm 4.2.1 Syntactically independent variable decomposition

```
1  Input: Hybrid automaton A with variables X := x_1,...,x_n
2  Output: A set of sets of variable indices
3
4  Graph g
5  // add node for each variable
6  for(x_i ∈ X){
7    g.addNode(x_i)
8  }
9
10 // for each location in A
11 foreach(l ∈ A.getLocations()){
12   foreach(flow ∈ l.getFlows()){
13     foreach((x_i,x_j) ∈ flow.getVariables()){
14       g.addEdge(x_i,x_j)
15     }
16   }
17   foreach(inv ∈ l.getInvariants()){
18     foreach((x_i,x_j) ∈ inv.getVariables()){
19       g.addEdge(x_i,x_j)
20     }
21   }
22 }
23
24 // for each transition in A
25 foreach(t ∈ A.getTransitions()){
26   foreach(guard ∈ t.getGuards()){
27     foreach((x_i,x_j) ∈ guard.getVariables()){
28       g.addEdge(x_i,x_j)
29     }
30   }
31   foreach(reset ∈ t.getResets()){
32     foreach((x_i,x_j) ∈ reset.getVariables()){
33       g.addEdge(x_i,x_j)
34     }
35   }
36 }
37
38 //for each bad state
39 foreach(state ∈ A.getBadStates()){
40   foreach(constraint ∈ state.getConditions()){
41     foreach((x_i,x_j) ∈ constraint.getVariables()){
42       g.addEdge(x_i,x_j)
43     }
44   }
45 }
46
47 // compute connected components
48 return  g.connected_components()
```

DecisionEntity
`getRepresentationForSubspace(Location,index) : representation`
`getRepresentationForLocation(Location) : representation`
`getRepresentationForAutomaton(Automaton) : representation`
`getSubpaceDecomposition(Automaton) : vector<vector<int»`
`isTimedSubspace(Location,index) : bool`
`isTimedLocation(Location) : bool`
`isTimedAutomaton(Automaton) : bool`
`isDiscreteSubspace(Location,index) : bool`

Figure 4.3: `DecisionEntity` class.

the only current available reachability worker which is, as stated earlier, too inflexible to support such an approach.

The approach is a refinement of the scenario-based design introduced in [FR09]. The idea of that design is to have a collection of algorithms called scenarios implementing a common interface that are suitable to analyze specific types of automata. Each scenario has to offer functionality to compute continuous and discrete behavior as well as functionality to maintain a list of passed and waiting states, similar to the task queue in HyDRA (see Figure 4.1). In fact, the current reachability worker can be seen as a scenario for linear hybrid automata, currently the only scenario implemented in the HyDRA toolbox.

In our approach we introduce the idea of a *context* that is a refinement of the automaton based scenario to a location based context. Based on the location's composition of classified subspaces a context is provided by a builder pattern that combines an algorithm (e.g. flowpipe analysis or zone elapsing) with specialized operations for continuous dynamics and discrete jumps of the respective subspace classification.

4.3.1 General Structure

Recall that HyDRA's architecture uses a thread pool pattern, in which workers repeatedly process and create tasks until no tasks are left. These workers first get a task from the task queue followed by a reachability computation for the state given by the task after which they will create new tasks based on the jump successors computed during analysis. In the current reachability worker's implementation the entire analysis algorithm is inside the `computeReachability()` method (see Figure 4.1).

The new context-based worker uses *switchable* contexts such that a worker can now perform reachability analysis based on the problem instance at hand. This worker is based on the observation that a lot of reachability algorithms for hybrid automata perform the same steps (invariant checks, guard intersections) and only differ in how the steps are implemented for the problem instance.

In order to make contexts interchangeable they all have to provide common functionality to perform reachability analysis. This functionality includes computation of a first segment, intersection with invariants, guards and bad states as well as continuous evolution and creation of jump successors. Consequently, we introduce an interface that all contexts have to implement to provide the methods needed to be used in the workers reachability algorithm (see Figure 4.4).

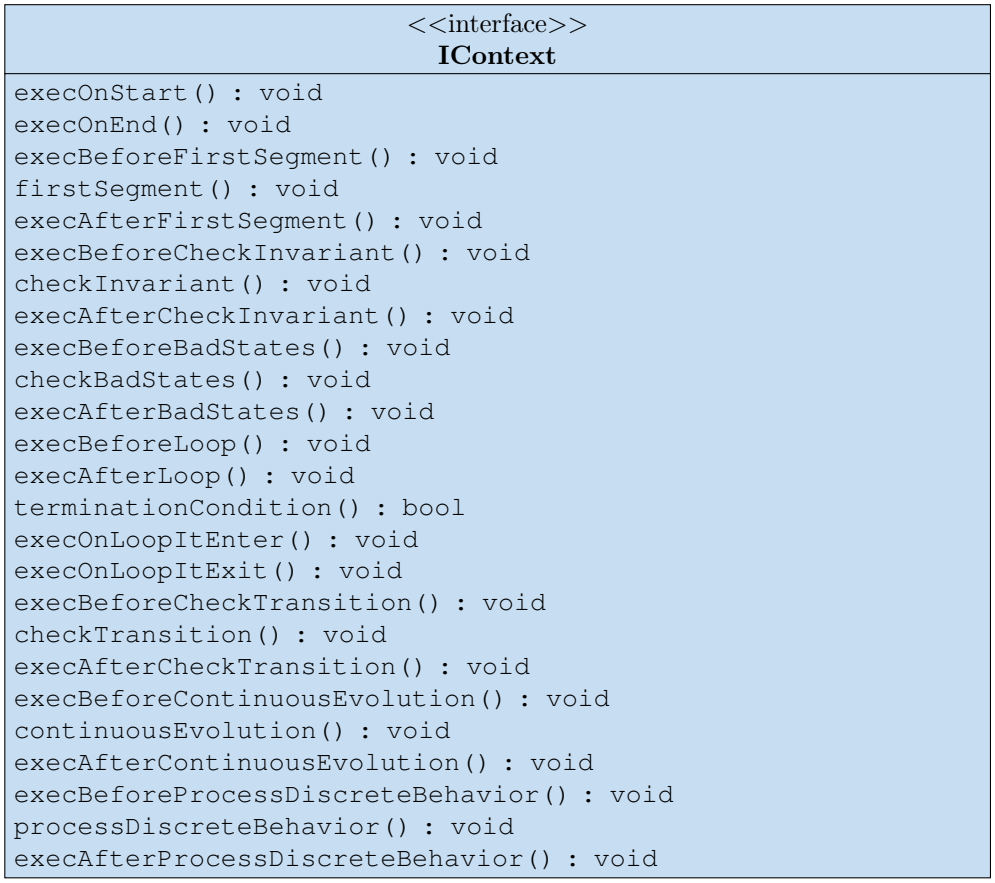


Figure 4.4: Context interface.

Additionally to these methods, the interface introduces methods prefixed by "exec" that provide *strategic entry points* for contexts extending other contexts. For example a flowpipe construction-based context that would like to perform a check for zeno behavior would extend a context implementing a default flowpipe algorithm and then use one of the "exec"-methods to "inject" the zeno checking code at the specific position, thus eliminating duplicate code and allowing for *rapid prototyping*.

If an extending context does not want to inject code at that position, it provides an empty implementation of that method. Consequently, as one can suspect, one does no longer have to program an algorithm. The algorithm is defined by the order the methods of the context are called in by the worker, hence one only has to define a context implementing these methods. In Section 4.5 we will see how we can implement a context for timed locations/automata with very little effort.

To get a better understanding of how the concept works we take a look at the algorithm used by the context-based worker as depicted in Algorithm 4.3.1. The methods that are not meant for code injection are highlighted by bold text and define the general reachability procedure. Note that none of the work is actually performed in the worker, instead it is handled by the context and its defined behavior.

The algorithm hands the task off to a context builder which is an entity that uses the decision entity of Section 4.2 to classify the task and return a context that is suitable for analyzing the state given in the task (line 2). This classification step also involves converting any unsuitable subspace representations to the representation suggested by the decision entity.

After obtaining its context, the algorithm computes the first segment of the flowpipe (line 7) followed by a check for intersection with the location's invariant (line 11) and automaton's bad states (line 15). Subsequently it will start computing continuous successor states while the termination condition has not been met (line 20).

In each iteration the algorithm checks for possible jump successors (line 24), applies the continuous dynamics to the variables (line 28) followed by another invariant and bad states check. When the termination condition is met and the loop terminates the collected jump successors are processed in line 45 to create follow-up tasks for the task queue.

In the next two sections of this chapter we are going to introduce implementations of two contexts the first one being the *default context*. As the name suggests it is intended to be the most general context. The second one will be a *timed context* that extends the default context to allow for faster analysis in timed locations/automata using zone elapsing.

On top of that, we will learn how a context copes with the different subspace types, i.e. decides which operation has to be used for a certain subspace.

4.4 Default Context

In this part of the thesis we are going to implement a context that is able to perform general flowpipe construction-based reachability analysis for a linear hybrid automaton. It will also be capable of performing this computation in *syntactically independent subspaces* w.r.t. the individual subspace classifications. As it is supposed to serve as the most general context it solely implements the core methods of the context interface (bold in Algorithm 4.3.1) and none of the "exec" methods.

The default context employs a *handler-based* approach to perform the different operations required for reachability analysis by assigning a set of possibly stateful *handlers* to each subspace that will perform the operations for the specific subspace.

To give an idea of what a handler may look like we present pseudo code for an invariant intersection handler for a linear classified subspace.

The given invariant handler is initialized with a reference to the current state of the automaton (line 3) and an index of the subspace it is supposed to work on (line 4). When the `handle()` method is called the intersection of the subspace at the specified index with the invariant for that subspace is performed (line 9) and the result is stored in boolean variable (line 5) which can be read at a later point. If we have to perform another invariant check we just call the `handle()` method again to update the boolean variable. Accordingly, a handler for invariant intersection in a timed classified subspace would perform intersection with a constraint as described in Algorithm 3.3.1 because the underlying subspace can be represented by a difference bound matrix.

Consequently, the default context maintains a list of handlers for each operation and when one of the operations has to be performed calls every handler and thereafter processes the results. An exception to the rule are handlers for guard intersection,

Algorithm 4.3.1 computeReachability in a context based worker

```
computeReachability{
  context = ContextBuilder.getContext(task)

  context.execOnStart()

  context.execBeforeFirstSegment()
  context.firstSegment()
  context.execAfterFirstSegment()

  context.execBeforeCheckInvariant()
  context.checkInvariant()
  context.execAfterCheckInvariant()

  context.execBeforeCheckBadStates()
  context.checkBadStates()
  context.execAfterCheckBadStates()

  context.execBeforeLoop()

  while(!context.terminationCondition()){
    context.execOnLoopItEnter();

    context.execBeforeCheckTransition()
    context.checkTransition()
    context.execAfterCheckTransition()

    context.execBeforeContinuousEvolution()
    context.continuousEvolution()
    context.execAfterContinuousEvolution()

    context.execBeforeCheckInvariant()
    context.checkInvariant()
    context.execAfterCheckInvariant()

    context.execBeforeCheckBadStates()
    context.checkBadStates()
    context.execAfterCheckBadStates()

    context.execOnLoopItExit()
  }

  context.execAfterLoop()

  context.execBeforeProcessDiscreteBehavior()
  context.processDiscreteBehavior()
  context.execAfterProcessDiscreteBehavior()

  context.execOnEnd()
}
```

Algorithm 4.4.0 LinearInvariantHandler

```
1  class LinearInvariantHandler {
2
3    State state;
4    int index;
5    bool satisfies;
6
7    void handle(){
8      List<Halfspace> inv = state.getInvariant(index);
9      satisfies = state.getSubspace(index).satisfiesHalfSpaces(inv);
10   }
11
12   bool satisfiesInvariant(){
13     return satisfies;
14   }
15 }
```

which are stored in a map rather than a list. The reason for this is that a location in general can have more than one outgoing transition and each of these transitions has its own guards and therefore deserves its own list of handlers (see Figure 4.5). To support interchangeability of the handlers they implement an according interface for each operation.

DefaultContext
`List<IFirstSegmentHandler> firstSegmentHandlers`
`List<IInvariantHandler> invariantHandlers`
`List<IBadStateHandler> badStateHandlers`
`List<IContinuousEvolutionHandler> continuousEvolutionHandlers`
`Map<Transition,List<IGuardHandler» transitionHandlerMap`

Figure 4.5: DefaultContext.

In the following section we are going to explore how the default context performs its operations by calling the respective handlers. Note that its the task of the decision entity to decide which handlers are constructed based on the subspace classification it decides on.

4.4.1 Algorithm

As mentioned before, the default context implements all of the core methods that are printed in boldface in Algorithm 4.3.1. As most of the methods are implemented by only calling the respective handlers we are only going to investigate testing the termination condition and invariant intersection as an example for result processing.

Termination condition

The termination condition in a default context involves checking a flag for early termination of the loop as well as whether the current time has passed the time horizon

for this location, i.e. the flowpipe has been fully computed for the given time horizon.

Algorithm 4.4.1 terminationCondition()

```
1  bool terminationCondition(){
2    return endLoop || (currentTime > timeHorizon);
3  }
```

Invariant intersection

Invariant intersection consists of a call to each handler (line 5) while simultaneously evaluating the result (line 6). If one of the handlers returns a violation of its invariant, the whole states intersection with the invariant is empty.

Furthermore, each handler is able to request its removal from the list of handlers and is removed when the computation is done (line 13). This is especially useful if a subspace is classified as a discrete subspace. In case of a discrete subspace the intersection with an invariant only has to be performed *once* as the valuations in that subspace are persistent.

Algorithm 4.4.1 checkInvariant()

```
1  void checkInvariant(){
2    bool satisfied = true;
3    vector<int> toDelete;
4    for(int i=0; i < invariantHandlers.size();i++){
5      invariantHandlers[i].handle();
6      satisfied &= invariantHandlers[i].satisfiesInvariant();
7
8      if(invariantHandlers[i].deleteRequested()){
9          toDelete.push_back(i);
10     }
11   }
12
13   invariantHandlers.erase(toDelete);
14
15 }
```

Transition checks are performed similarly. For each transition the respective list of handlers is called and the result evaluated. If all subspaces satisfy their transition guards the respective transition is enabled and the state saved for later task creation. If a handler requests its removal it is removed from its list.

A special case arises when a discrete subspace dissatisfies its transition guard. In this case the transition can *never* be taken and the whole list of handlers is removed to avoid future checks. Consequently, it is a wise choice to check discrete subspaces first and if they invalidate the transition guard, we can disregard that transition for the rest of flowpipe construction which can save thousands of intersection operations.

Adding a sort index to handlers that imposes an *order* on handlers (and therefore implicitly on subspaces) is an interesting idea, but raises the question of what ordering

may be the best and some ideas will be suggested in Chapter 6.

4.5 Timed Context

The main difference between a timed and a default context lies in the way continuous evolution of the variables is computed. In a default context, continuous evolution for linear subspaces is performed by applying a linear transformation as described in Definition 4.2.2 and for timed subspaces by performing shift operations as defined in Section 3.3.3 to let the clocks tick. A timed context on the other hand is used if the automaton's current location is a timed location where all subspaces are timed subspaces, hence we aim at computing the whole reachable state space in a location at once using zone elapsing.

At first, this seems like an easy task. We can just implement a continuous evolution handler that performs *zone elapsing* instead of *clock shifting*. Technically, this works perfectly and precise as long as we do not decompose the state space into syntactically independent subspaces. If the state space is decomposed, enormous over-approximation errors are introduced that make the result unusable. This is due to the fact that clocks are directly tied to the passage of time and the default context respects this by only allowing clock ticks to maintain *synchronicity* with the other subspaces. Any over-approximation error by decomposition is therefore limited by the size of the time step used for flowpipe construction.

In a timed context where any time span can be elapsed in a single step this synchronicity is lost and *arbitrary large errors* are introduced. Consider the following simple example of timed location and state on entrance in Figure 4.6.

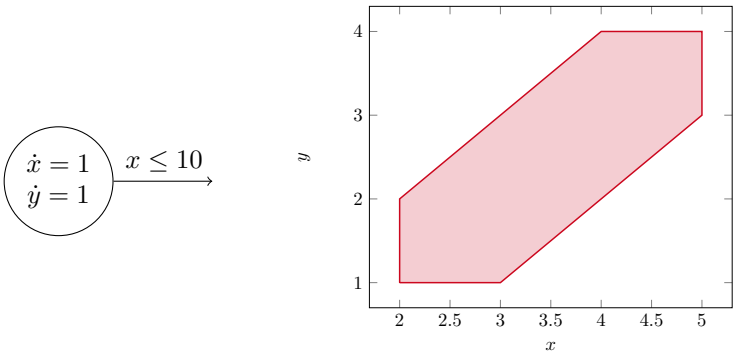


Figure 4.6: Time location and state on entrance.

Elapsing the zone and intersecting it with the transition guard yields the zone in Figure 4.7.

Now consider the same example with a decomposition into subspaces (Figure 4.8). As x and y do not depend on each other they are in independent subspaces. On the left hand side the entrance state is projected to the individual subspaces. On the right hand side we see the result after first elapsing the subspace zones by removing their upper bounds followed by the guard intersection with $x \leq 10$ in the x subspace. As we can see, the x subspace reaches from 2 to 10 as it would in the non decomposed

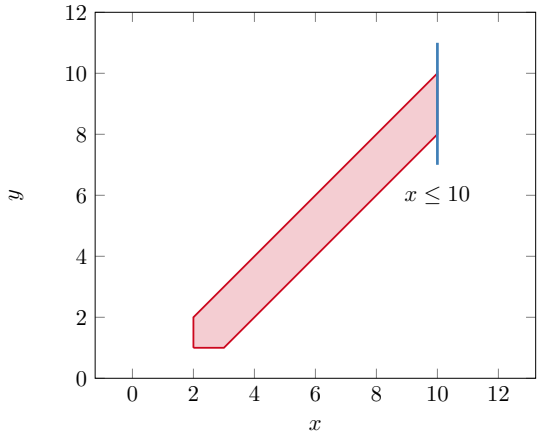


Figure 4.7: Zone from Figure 4.6 elapsed and intersected with $x \leq 10$.

setting. The y subspace however was not constrained by any guard and hence became infinite introducing an *infinite over-approximation error*.

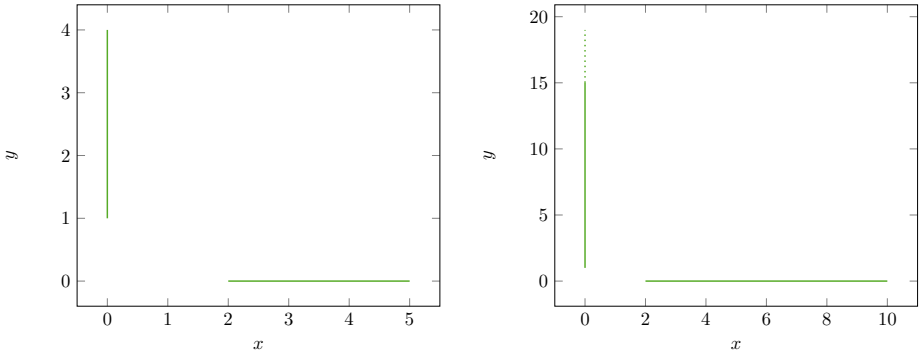


Figure 4.8: Reachability in a decomposed setting.

Naturally, this result is unusable because clocks are directly tied to the passage of time and it cannot be that one clock elapsed 5 to 8 seconds while the other elapsed an infinite amount of time. Therefore, simply replacing a continuous evolution handler performing a clock shift with a handler performing zone elapsing is not enough to compute a valid reachable state set in one step - at least not if variable set separation is used.

This leads to the idea of a *handshake*, where clocks agree on a time frame that has passed since location entrance to maintain *synchronicity*.

4.5.1 Handshaking

As we have seen in the previous section, the use of zone elapsing in a decomposed timed context introduces arbitrary large over-approximation errors that render any

reachability computation almost useless. In this section, we present a *handshake procedure* to solve this issue.

In this handshake procedure all clocks have to agree on a time span that may have passed between a state before and after an operation happened. Reconsider Figure 4.8 and that x is constrained whereas y is not. Consequently, in x's universe a time of 5 to 8 seconds has passed before the transition guard was invalidated while in y's universe an infinite time span has passed. Consequently, x has to tell y that it has to restrict itself to a time span that is y's starting span (1 to 4 in Figure 4.8) plus the time that x says has passed before the guard was invalidated, which is 5 to 8 seconds. As a result, the valid range of y values should be between 1 and 9 seconds.

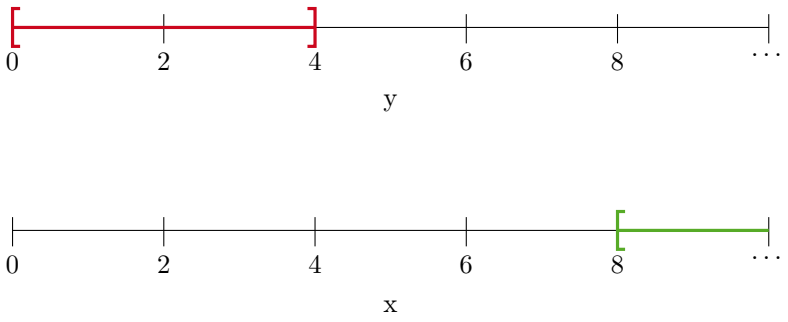
Note that the main problem in a handshake procedure is that clocks can be reset to 0. Therefore it is not enough to take the clocks valuations, intersect them and then somehow compute the new elapsed time span for each clock, since the absolute valuations of the clock are not the point of interest. Its the *relative time* that has elapsed since the start of the operation that has to be considered when handshaking. For example, x has to tell y that 5 to 8 seconds have passed since the earliest possible start at 2, not that its time is between 2 and 10 after the operation because that information has no value for y, since in y's universe time started at 1.

Consequently, for each operation there is a time span described by each clock before that operation and one time span described by the clock after that operation. The time span after an operation can either become larger in case of continuous evolution, or shorter in case of invariant/guard intersection since these operations cut invalid clock values. As all clocks evolve at the same constant rate of 1 no handshake is needed after continuous evolution.

After intersection with a guard or invariant there are basically two cases that have to be considered when performing a handshake, i.e. whether or not the elapsed time intersects. We will see that both of these cases come down to finding the *tightest overlapping time span* over all timed subspaces.

We start by investigating the case where time spans do not intersect.

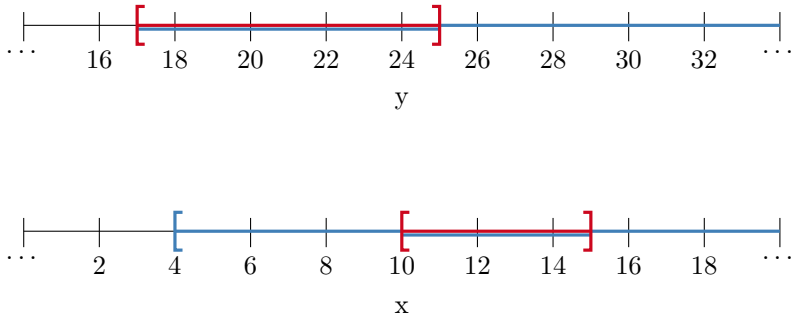
Consider two clocks x and y that both started at 0 and are about to check for a transition with guard $x \geq 8 \wedge y \leq 4$. In a decomposed setting, this yields the following subspace valuations for the transition.



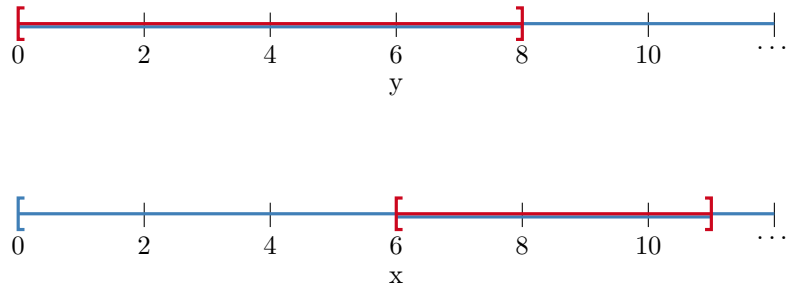
Without a handshake, both subspaces would signal that they satisfy their part of the guard and hence the transition would be taken. However, taking a closer look we can see that y states that an amount anywhere between [0,4] has passed before the transition was taken whereas x states that at least [8,∞] time units must have passed before its guard is satisfied. Consequently, a handshake procedure fails at that point

as both clocks did not manage to agree on a time span for the transition and hence it is not taken.

Now consider an example where both time spans intersect and agreement is possible. Let x and y be about to check a transition with guard $x \geq 10 \wedge x \leq 15 \wedge y \leq 25$. Furthermore, let the valuations after elapsing be $x = [4,\infty]$ and $y = [17,\infty]$. After intersecting each clock with its transition guard we obtain the following situation. Note that the axes describe different points in time. The blue intervals are a visual indicator of the valuation before the intersection, red intervals depict the situation after intersection.



The guard satisfying sections of time are $x = [10,15]$ and $y = [17,25]$. As we can see, intersecting these two absolute intervals would yield an empty intersection and consequently a transition would not be taken. However, as we stated earlier, the absolute time intervals are irrelevant because we have to consider the intervals since the start of x and y, which is $x = 4$ and $y = 17$. Consequently we shift the valuation before guard intersection to the origin of time (i.e. 0) and shift the guard satisfying intervals accordingly. Note that this is possible because according to Definition 2.4.1 all clock valuations have to be ≥ 0 so no clock interval can be of the form $[-\infty,c_1]$ which would render a shift to 0 impossible. Shifting the clocks as described above yields the following *normalized view* of passed time.



This view tells us that in y's universe [0,8] time units may have passed before the transition was taken whereas an amount of at least [6,10] time units may have passed in x's universe. The intersecting time interval of [6,8] is the agreed time span that has to be passed in both subspaces before the transition is taken. Consequently, a valid time span for the transition is $x = [4+6,4+8] = [10,12]$ and $y = [17+6,17+8] = [23,25]$ and the subspaces are constraint accordingly.

Recall that the original goal of this section was to come up with a timed context that is able to analyze *timed locations* by zone elapsing and while simply using an elapse handler for continuous evolution works when we do not decompose the automaton, it failed in a decomposed setting due to the lack of handshake.

With a handshake at our disposal a timed context can be implemented by performing zone elapsing for continuous evolution and following up each operation performed during analysis (e.g. guard/invariant intersection) with our handshake method described above to ensure that the result of each operation is valid. Consequently, a timed context is an extension of the default context where each of the `execAfter⟨Operation⟩`-methods is implemented by performing a handshake between the subspaces (see e.g. line 6 in Algorithm 4.5.1). Furthermore, the termination condition is replaced with a simple flag (line 3 and line 26) that is turned to true after the first iteration (line 33) since the whole reachable zone is computed in one step. Since the transition check is performed at the start of the loop, we have to redo it at the end of loop when the zone has been elapsed (line 31).

As stated earlier, it is the task of the decision entity to decide on which handlers to use, so using a zone elapsing-based continuous evolution handler is done in the default context based on the fact that the decision entity determined that we are in a timed location.

In the following section we are going to test our implementation against common benchmarks.

Algorithm 4.5.1 TimedContext

```
1  class TimedContext extends DefaultContext {
2
3    bool done = false;
4
5    void execAfterFirstSegment(){
6      performHandShake(beforeState, currentState);
7    }
8
9    void execAfterCheckInvariant(){
10     performHandShake(beforeState, currentState);
11   }
12
13   void execAfterCheckBadStates(){
14     performHandShake(beforeState, currentState);
15   }
16
17   void execAfterCheckTransitions(){
18     performHandShake(beforeState, currentState);
19   }
20
21   void execAfterContinuousEvolution(){
22     performHandShake(beforeState, currentState);
23   }
24
25   bool terminationCondition(){
26     return done;
27   }
28
29   void execOnLoopItExit(){
30     execBeforeCheckTransition();
31     checkTransition();
32     execAfterCheckTransition();
33     done = true;
34   }
35 }
```

Chapter 5

Benchmarks

In this section of the thesis we evaluate our implementation against common benchmarks, i.e. elaborate versions of the *thermostat*, *leaking tank* and *two tanks* example. Furthermore, we add our *cooking automaton* from Section 2.4 as an example for a timed automaton to show the drastic effect zone elapsing-based reachability analysis can have by using the aforementioned timed context.

5.1 Benchmark Suite

The majority of this *benchmark suit* is taken from [SNÁ17] and are common examples in hybrid reachability analysis. For the purpose of their work, the authors altered these automata to model *programmable logic controllers (PLCs)* such that they have a lot more variables, i.e. clocks and discrete variables to simulate PLC cycles. Furthermore, as actuators, sensors and controller of a device controlled by a PLC are somewhat separated by definition it is to be expected that such a system decomposes whereas standard examples in literature tend not to as they abstract from hardware.

5.1.1 Thermostat

This example models the thermostat from Example 2.1.1 augmented by a PLC controller. The initial temperature is $20°C$ and the heater is turned on. The controller then tries to maintain a temperature between $16°C$ and $23°C$ by switching off the heat at $22°C$ and turning it back on if below $18°C$. The PLC controller has an internal cycle time of 0.5 seconds and a global time horizon of 10 seconds is observed.

The model augmented by a PLC controller consists out of a total 8 variables: 5 discrete variables and 2 clocks for the controller as well as one variable that stores the current temperature.

The results of a reachability computation for this example can be found in Figure 5.1.

5.1.2 Leaking Tank

A leaking tank is a water filled tank that has a constant outflow. To keep the tank from becoming empty it can be refilled with a constant inflow from an unlimited resource of water. A PLC controller tries to maintain a water level between 6 and 12.

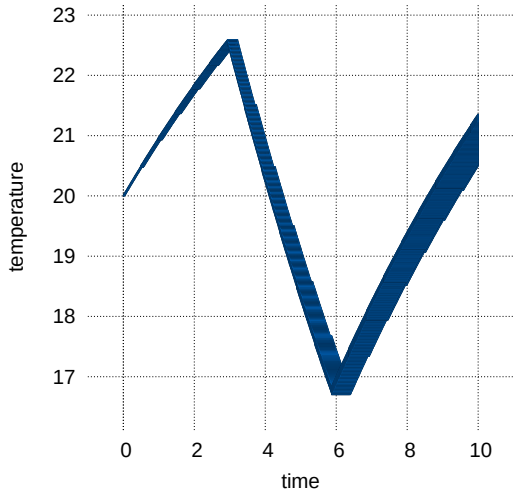


Figure 5.1: Thermostat example (box representation).

When the water level is low the PLC triggers refilling of the tank and stops refilling when it is full. Additionally, this examples models a user who can decide to manually refill the tank until it is full. The PLC controller has an internal cycle time of 2 seconds and a global time horizon of 40 seconds is observed.

The model consists out of a total 12 variables: 9 discrete variables and 2 clocks for the controller as well as one variable that stores the current water level.

The results of a reachability computation for this example can be found in Figure 5.2.

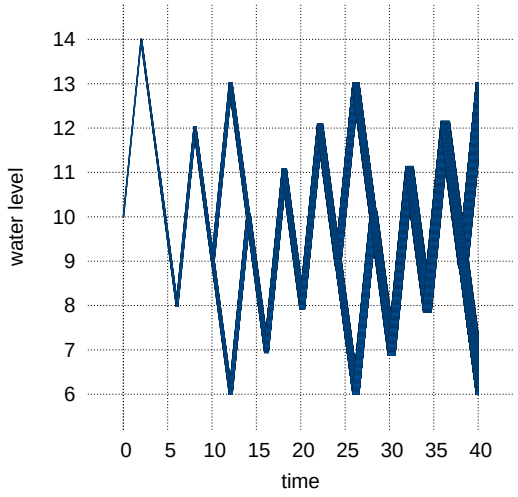


Figure 5.2: Leaking tank example (box representation).

5.1.3 Two Tanks

The two tank example models the water level of two water tanks that are connected via pipes. Both tanks have a constant in- and outflow such that the amount of water that flows out the first tank equals the amount of water that flows into the second tank and vice versa. A PLC controller tries to maintain a water level between 8 and 32 and as in the leaking tank example a user is modeled who can decide to pump water from one tank to another. The PLC controller has an internal cycle time of 1 second and a global time horizon of 20 seconds is observed.

The model consists out of a total 22 variables: 17 discrete variables and 3 clocks for the controller as well as 2 variables that store the current water levels in the respective tanks.

The results of a reachability computation for this example can be found in Figure 5.3 and Figure 5.4.

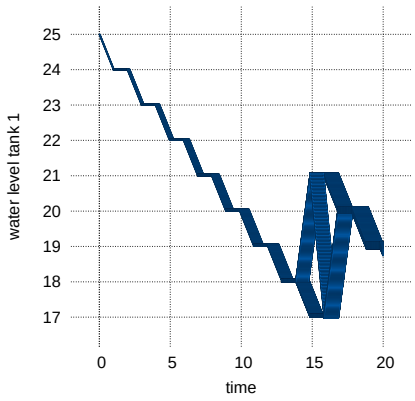


Figure 5.3: Two tanks example first tank (box representation).

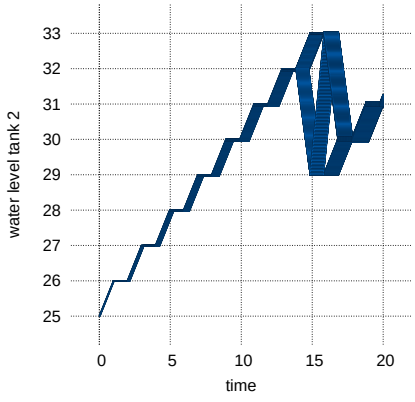


Figure 5.4: Two tanks example second tank (box representation).

5.1.4 Cooking Automaton

In this example we observe the cooking automaton from Example 2.4.1. The example has been altered such that all guards and invariants in this automaton are given in seconds rather than minutes to pose a more significant challenge. The automaton is observed for a total amount of 7200 seconds (2 hours).

The results of a reachability computation for this example can be found in Figure 5.5.

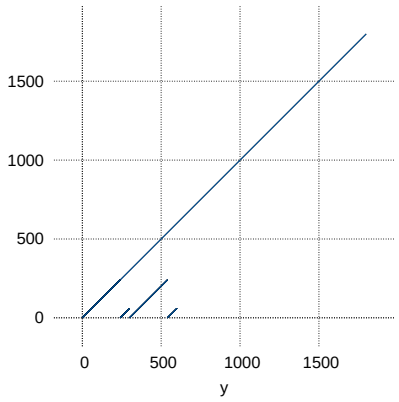


Figure 5.5: Cooking automaton example (DBM representation).

5.2 Experimental Results

All experiments were carried out on an AMD Ryzen 5 1600 3.2GHz six-core CPU with 8GB RAM. A time step of $\delta = 0.01$ and unlimited jump depth were used. The times are obtained as an average of 10 subsequent runs. We distinguish 5 different types of run configurations:

Original In this run configuration we performed reachability analysis as it is performed by the current implementation of HyDRA to serve as a point of reference for the enhancements of this thesis. Consequently, *none* of the improvements of this thesis are employed.

Decider In this run we use the decision entity, but do not chose to decompose the automaton. Hence, in this run a classification of a *non decomposed automaton* is performed. Consequently, these runs differ from "original" runs by using the timed context if a timed automaton is analyzed.

Decompose In this run we use the decision entity to decompose the state space into syntactically independent subspaces, but we do not use it to classify the subspaces. Consequently, none of the classification specific enhancements, like e.g. switching the

representation to a difference bound matrix in case of a timed subspace, are performed. Solely the effect of *decomposing* an automaton is benchmarked here.

Decider + decompose In this run we combine the "decider"- and "decompose" run configuration. Consequently, the state space is *decomposed* into syntactically independent subspaces and the *classification* of these subspaces is used to switch to difference bound matrices where timed subspaces occur.

Context Switch (decider + decompose) In this experimental run we combine the "decider + decompose" run configuration with the possibility to *switch* from a default context to a timed context in case we enter a location where all subspaces are classified as timed. Note that if this run is performed on a non timed automaton and has an effect this means that at least one of the variables that is not a clock by definition behaves like a clock in at least one of the locations. This can happen e.g. if a linear flow variable that is only compared against constants happens to have a flow of 1 in a location and is not reset on any outgoing transition (or reset to 0).

We label this run as experimental, because it can have an arbitrary large over-approximation error when converting the representation of a variable set that previously did not behave like a clock to a difference bound matrix that assumes clocks.

Performing the above runs on all 4 automata yields the benchmark results given in Table 5.1. The table presents both the *absolute run time* in seconds averaged over 10 consecutive runs and the *relative speed-up* compared to the "original" run computed as $t_{new}/t_{original}$. The runs are performed for the imprecise but very fast box and the more precise but slower support function.

Furthermore, the speed-up table cells are *color coded*. A yellow table cell states there is no significant speed-up ($\leq 5\%$ faster/slower). A red table cell states that we are significantly slower ($\geq 5\%$) and a green table cell states that we are significantly faster ($\geq 5\%$).

Before we dive into individual results there are some global remarks regarding the benchmark results:

- The cooking automaton differs from the other automaton in that it is the only timed automaton of the benchmark. Naturally, it has *extreme speed-ups* in any run where the decision entity is used to employ difference bound matrices. The speed-up is especially large when one step reachability via zone elapsing is performed.
- On the other hand, decider runs gain *no speed up* for the other automata as it has literally no effect on them.
- We stated that an automaton is complexer the more variables it has. We can see this looking at the original run times where thermostat $<$ leaking tank $<$ two tanks holds for the run times.
- "decider + decompose" runs are *always faster* than "original" runs.
- Decomposing the state space is always faster for support functions, but not always for boxes.

		original	decider	decompose	decider+decompose	context switch
Thermostat						
Box	Average	0,175	0,177	0,173	0,148	0,237
	Speed-up	1	0,992	1,010	1,181	0,739
Support Function	Average	8,356	8,288	0,527	0,186	0,269
	Speed-up	1	1,008	15,845	44,790	30,971
Leaking tank						
Box	Average	1,683	1,666	2,005	1,267	0,888
	Speed-up	1	1,010	0,839	1,328	1,894
Support Function	Average	151,118	150,833	9,357	2,538	1,941
	Speed-up	1	1,001	16,149	59,540	77,839
Two tanks						
Box	Average	11,357	11,401	2,538	1,729	1,547
	Speed-up	1	0,996	4,475	6,568	7,340
Support Function	Average	854,694	844,282	7,641	2,323	2,266
	Speed-up	1	1,012	111,846	367,855	377,110
Cooking						
Box	Average	15,903	0,315	29,905	0,157	0,156
	Speed-up	1	50,485	0,531	101,082	101,670
Support Function	Average	322,218	0,318	249,338	0,155	0,156
	Speed-up	1	1011,447	1,292	2066,082	2056,168

Table 5.1: Benchmark results.

- The individual results heavily depend on the context and the relative speed-ups range from 0,53 (about 50% slower) to 2066,082741 (about 200.000% faster).

The first thing that catches a readers eye is that a decompose run is almost always *slower* (or equal) than an "original" run when using boxes. The reason for this behavior is the union operation. When taking a transition, usually multiple states intersect the transitions guard. An algorithm could enqueue all of them as individual tasks, or *aggregate* them using the union operation and in fact HyDRA offers the option to choose whether to aggregate or not. The union operator for boxes is extremely fast and it is faster to unite e.g. 15 6-dimensional boxes than to unite 90 1-dimensional boxes. As the box is rather fast in general, performance gains at other operations (intersection) do not outweigh the *performance loss performing union*. One may notice that this effect is especially significant in the leaking tank example. This is because the leaking tank example has the longest PLC cycle time, so naturally a transition may be enabled the longest in this example, generating more and more successor states that have to be aggregated.

Another rather interesting result is that in the thermostat example the context switch is rather slow. This is due to the fact that when the automaton enters a timed location, but spends no time in it, we still have to perform handshake operations for the individual operations and subspaces. These handshakes involve multiple intersection that are pretty costly. This raises the question whether or not there is a way to tell of what the "expected" time spent in a location is to *avoid switching* to the timed context and handshakes if we spent no time in a location. The idea that it may be possible to have an *expectation* of time spent stems from the fact that some variables are clock variables in the entire automaton, so if an invariant of a location states that a clock has to have the same value as the clock has on entrance that suggests that we do not spend any time in that location.

On the contrary, the context switch runs for leaking tank and two tanks is faster because we actually spent time in a timed location.

Regarding the cooking automaton, which is a timed automaton, we can see that runs that do not involve the timed context (no decider runs) are $1000 - 2000$ times slower than runs using a decider and hence the timed context using one step reachability. Furthermore, for the cooking automaton a timed context using decomposition is 100% faster than without decomposition. The fact that a "context switch" run is slightly slower than a "decider + decompose" run can only be explained by operating system load and additional boolean checks.

Overall, the results suggest that run times of flowpipe construction-based reachability analysis are extremely *dependent on the context* they run in, so having a more and more *intelligent* decision entity is a practicable way to reduce running times.

Chapter 6

Conclusion

In this thesis the effects of *context-dependent reachability analysis* in hybrid system have been explored. A *decision entity* was developed that is able to identify various properties of a hybrid automaton and still leaves room for a lot of experiments and discoveries.

Furthermore, a context-dependent framework was developed in the HyDRA toolbox that supports reachability analysis based on *variable set separation* as well as *rapid prototyping*. It has been shown that with this framework a timed context for analyzing timed locations/automata can be implemented with very little effort.

To demonstrate the capabilities of such a framework as well as to emphasize the result achievable by an intelligent, decomposed analysis we enriched HyPro's various representations by an efficient *difference bound matrix* that exploits the special properties of clock variables in a timed scenario to both save time and memory.

It has been shown that a speed-up of up to 2000 times is possible, growing with the size of the problem instance. It has also been shown, that run times heavily depend on the context defined by a location, its initial valuation and variable set separation. Consequently it is to be a expected that a more and more *intelligent* decision entity with a more elaborate subspace classification as well as more static information about the automaton may lead to faster run times.

6.1 Future Work

This thesis is a gateway for a lot of enhancements and discoveries to come as it lays a *powerful framework* for variable set separation-based and intelligent flowpipe construction. Along our way, we encountered a lot of questions and curiosities that exceeded the scope of the thesis, although I would have loved to take a peak at them. Consequently, I pose these suggestions as future work.

Regarding HyPro and DBMs

As DBMs where only one part of the thesis, they are not at 100% of their potential. Especially functionality that is part of the geometric object interface, i.e. conversion to other representations and operations not designed for DBMs are implemented via conversion to hyperplane polytopes followed by performing the corresponding conversion/operation and then converting back. This is not very performant, so a specialized implementation of these conversions/operations is desirable. Furthermore, papers hint at the fact that *sparsity* of DBMs may be exploitable.

Regarding HyDRA

As a decision entity becomes more intelligent HyDRA should offer more functionality for different types of automata. For example, *rectangular automata* are not supported by HyPro/HyDRA. With additional constructs, *additional handlers and contexts* should be developed. Hopefully, this leads to a *rich environment* of contexts and handlers in the future.

Another thing that should be investigated on is that timed subspaces should always be analyzable using zone elapsing regardless of whether the other subspaces are timed or not. This is due to the fact that the elapsed subspace should always be restrictable to the current time step. The question is whether or not this actually saves time or not.

The benchmark suggests that it is not always the best choice to decompose the state space of the automaton as much as possible. There are various ideas on how to condense variable sets to allow for faster computation. Some of the ideas include but are not limited to:

- *Grouping of clocks.* The attentive reader may have noticed that our decomposition algorithm puts each clock into its own subspace due to the fact that difference constraints on clocks are disallowed in a general timed automaton.
- *Grouping of discrete variables.*
- *Grouping of variables based on the underlying representation.* It may be the case that a coarser separation is advisable for a box (as unions are fast at high dimensionalities) but is discouraged for hyperplane polytopes where unions are particularly slow.

The current decomposition algorithm assumes that the dependency relation is *symmetric*, i.e. if x depends on y then y also depends on x. In general, that is not the case as x may not need y valuations to be computed, but y may need x valuations to be computed, e.g. $x = 2 \cdot t$ and $y = 2 \cdot x$. Thus, it may be possible to first compute the entire evolution of x and compute y on-the-fly as some kind of projection or transformation of x, opening up further optimization potential.

The benchmarks suggest that even a rough estimation of how much time is spent in a location could speed up computations. It should be investigated whether or not it is possible given certain circumstances.

Certain subspaces may be faster to analyze using certain representations, as DBMs for timed subspaces suggest. It should be investigated whether or not there are more instances where this is the case, i.e. boxes for discrete subspaces since they do not evolve over time. Furthermore, we have seen that some representations are more precise (e.g. support function) than others (e.g. box), so using a more precise representation in a subspace may counteract the over-approximation error introduced by subspace computations.

We have seen that *ordering* of handlers can play a crucial role regarding performance. For instance, it is advisable to check discrete subspaces first when considering a transition because a transition can never be enabled when a discrete subspace violates the transition guard. As more and more contexts and handlers are developed, we should have an eye for possible optimizations regarding handler ordering.

Furthermore, the location we are in should gain priority. The benchmarks, i.e. the context switch runs, suggest that there are significant performance enhancements when we disregard the overall properties of an automaton but restrict our view to the current location. For instance, when a variable behaves like a clock in a location it can be analyzed as a clock, regardless of whether or not it was a clock variable prior to this location. However, one should not forget about possible over-approximation errors and representation conversion costs.

Bibliography

[Ábr15] Erika Ábrahám. Lecture notes in modeling and analysis of hybrid systems, Summer Term 2015.

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

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. In *Theoretical Computer Science*, volume 126, pages 183 – 235. 1994.

[Bel57] Richard. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Bel97] Richard Bellman. *Introduction to Matrix Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[BK08] Christel Baier and Joost P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BY04] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, pages 87–124. Springer Berlin Heidelberg, 2004.

[CÁS13] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, pages 258–263. Springer Berlin Heidelberg, 2013.

[CS16] Xin Chen and Sriram Sankaranarayanan. Decomposed reachability analysis for nonlinear systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 13–24, 2016.

[Dil90] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer Berlin Heidelberg, 1990.

[FLGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer Berlin Heidelberg, 2011.

[FR09] Goran Frehse and Rajarshi Ray. Design principles for an extendable verification tool for hybrid systems. *IFAC Proceedings Volumes*, 42(17):244 – 249, 2009. 3rd IFAC Conference on Analysis and Design of Hybrid Systems.

[Frä99] Martin Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In *Computer Science Logic: 13th International Workshop, CSL'99 8th Annual Conference of the EACSL Madrid, Spain, September 20–25, 1999 Proceedings*, pages 126–139. Springer Berlin Heidelberg, 1999.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94 – 124, 1998.

[Hüt16] Dustin Hütter. *Adaptive dynamic reachability analysis for linear hybrid automata*. Master's thesis, RWTH Aachen University, September 2016.

[LG09] Colas Le Guernic. *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics*. Theses, Université Joseph-Fourier - Grenoble I, October 2009.

[ML03] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45(1):3–49, 2003.

[MSJM14] Ole G. Mouritsen, Klavs Styrbæk, Mariela Johansen, and Jonas D. Mouritsen. *Umami: Unlocking the Secrets of the Fifth Taste*. Arts and Traditions of the Table: Perspectives on Culinary History. Columbia University Press, 2014.

[Neu16] Johannes Neuhaus. *Development of a modular approach for hybrid systems reachability analysis*. Bachelor's thesis, RWTH Aachen University, September 2016.

[SÁ18a] Stefan Schupp and Erika Ábrahám. Efficient dynamic error reduction for hybrid systems reachability analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer International Publishing, 2018.

[SÁ18b] Stefan Schupp and Erika Ábrahám. Spread the work: Multi-threaded safety analysis for hybrid systems. To appear, 2018.

[SÁMK17] Stefan Schupp, Erika Ábrahám, Ibtissem Ben Makhlouf, and Stefan Kowalewski. Hypro: A c++ library of state set representations for hybrid systems reachability analysis. In *NASA Formal Methods*, pages 288–294. Springer International Publishing, 2017.

[SNÁ17] Stefan Schupp, Johanna Nellen, and Erika Ábrahám. Divide and conquer: Variable set separation in hybrid systems reachability analysis. In *Proceedings 15th Workshop on Quantitative Aspects of Programming Languages and Systems, QAPL@ETAPS 2017, Uppsala, Sweden, 23rd April 2017.*, pages 1–14, 2017.

[Sri12] Balaguru Srivathsan. *Abstractions for Timed Automata*. Theses, Université De Bordeaux, June 2012.

[Zie12] G.M. Ziegler. *Lectures on Polytopes*. Graduate Texts in Mathematics. Springer New York, 2012.