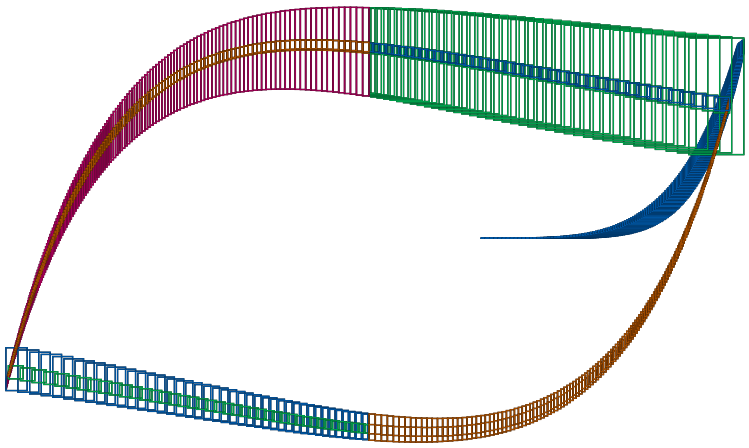
hybr2d Theory of Hybrid Systems Informatik 2 | RWTH AACHEN UNIVERSITY

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

BACHELOR OF SCIENCE THESIS

DEVELOPMENT OF A MODULAR APPROACH FOR HYBRID SYSTEMS REACHABILITY ANALYSIS

Johannes Neuhaus



Examiners:
Prof. Dr. Erika Ábrahám

Additional Advisor:
Prof. Dr. Jürgen Giesl

Aachen,
September 16, 2016

Abstract

Current methods for hybrid systems reachability analysis and verification, like bounded flowpipe-based reachability analysis, have been well-developed over the years resulting in various tool implementations. Optimization of those methods has been manifold. However, an exploitation of the discrete structure of a given system towards a parallel approach has been neglected so far. We present a parallelized implementation of a flowpipe-based reachability analysis algorithm as well as strategies for parallelized verification of hybrid systems. The implementation aims for extensibility by providing a library layer, which can act as an execution framework for parallel hybrid systems verification. The introduced improvements show a speedup of 65% in common benchmarks and runtime improvements of up to 360% in variants of those benchmarks whose structure has been optimized for this setup.

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

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

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

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

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

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

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

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

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Acknowledgements

I am grateful for the opportunity to write this bachelor thesis. Starting in the process of planning a new tool, I became part of a procedure which was incredibly interesting. Creating a new tool for hybrid systems reachability analysis required creativity, many discussions, as well as an in-depth understanding of hybrid systems. These fruitful discussions highly influenced what HyDRA is now. Additionally, it has been a great experience to try out new ideas like the multithreading approach. After all, the formal specification and proving of ideas, as well as the empirical examination of these ideas, was as new as interesting to me.

Many people supported me on the road to and during the writing of this thesis. First of all, I want to thank my parents for giving me the opportunity to study computer science. I also thank my friends, especially Simon, who proofread every single word, being nearly as precise as a commercial spell checker. By discussing the content of the thesis he also pointed out many potential improvements. I want to thank Fabian for many hours of fruitful discussion about multithreading and scaling problems.

Most important, I want to thank the people at the i2 chair. Thank you for having a look at dubious segmentation faults and all the explanations and discussions during the last five month! I especially want to thank Stefan Schupp, my advisor, who answered every single question I had during the last months, discussed and provided many ideas and supported me during the implementation in so many ways. I also want to thank Erika Ábrahám who made this work possible, as well as Jürgen Giesl, my additional advisor.

Contents

Chapter 1

Introduction

Many models in computer science like finite automata, Petri nets, and pushdown automata focus on the modeling of discrete models. These models are suitable for problems like parsing abstract languages or in the case of the Petri nets, they have been used as a tool for workflow management as well as a model for concurrency [VDA98, Mur89]. However, these can naturally not be used to represent processes which involve physics like the billiard game or the fall of a ball, without abstracting continuity. Being surrounded by physical processes, we are particularly interested in combining both - discrete and continuous behavior. Hybrid systems model exactly this class of problems. They can be used to describe physical processes like falling balls, the heating of a radiator or the way a billiard ball takes on a billiard table while combining it with discrete changes like the change of the direction of a billiard ball. Being able to model such a physical process using a hybrid system, we are interested

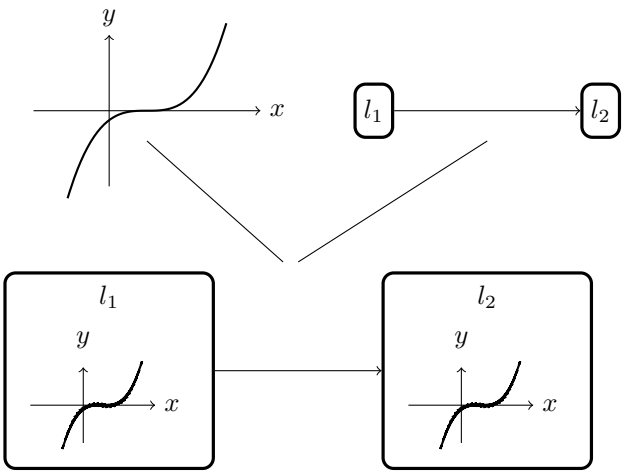


Figure 1.1: Depiction of the components of a hybrid system. A discrete system on the right and a continuous system on the left. Combined we get a hybrid system [AC15].

in the property of *safety*. We call a system safe when there is no possibility that a bad state occurs. For example, we will specify a model of a nuclear reactor, and we

call the system safe when there is no possibility that the temperature in the reactor rises to a specified value x. In this case, without knowing any details, a state of the system is intuitively described by the temperature of the reactor at time t. Then the system is safe when for no time t the temperature at time t is higher than x. However, testing for safety is not applicable for a reactor. Even if we were able to have test runs, we would never be completely sure whether we did not miss a test case revealing a modeling mistake. Thus, we are interested in *proving* safety of a model using formal methods. For the course of the thesis, we will model hybrid systems using a *hybrid automaton* and will call the computation of reachable states *reachability analysis*.

Before we introduce a new tool for hybrid systems reachability analysis in Chapter 3, we will study hybrid systems, especially hybrid automata, in Chapter 2 in more depth. As a new approach, this tool provides multithreading abilities which improve the runtime of common benchmarks by 60 percent, and 360 percent for benchmarks whose model has been optimized for this purpose, as we will show in Chapter 4. As currently most common benchmarks for hybrid systems reachability analysis and verification do not reveal much potential for the usage of multiple threads, we introduce strategies for these types of models in Chapter 3.4.

Chapter 2

Preliminaries

To understand the requirements for a modular verification task of hybrid systems, we need a formal model of hybrid systems. There are several ways to do verification and reachability analysis. Currently there exist several approaches towards the verification of a hybrid system. However, there are classes of hybrid systems which are known to be undecidable [HKPV98] which makes a computation of all reachable states impossible. For the class of linear hybrid systems - where all constraints and activities can be expressed as linear predicates - this problem is semi-decidable for overapproximated state sets and well-studied [LG09].

2.1 Hybrid Systems

Hybrid systems are systems which combine discrete and continuous behavior. The continuous behavior gives the opportunity to express physical processes which are for example continuous in time. Hybrid systems can be modeled using hybrid automata.

Definition 2.1.1 (Hybrid Automaton [ACH+95, LG09, ACHH93, AC15])**.** *A hybrid automaton H is a tuple*

$$H = (Loc, Var, Lab, Inv, Act, Trans, Init)$$

Locations: *Loc is a finite set of* locations*. Locations are the nodes of a directed graph. The edges are given by Trans.*

Variables: *Var is a finite set of real-valued variables. A valuation v for a variable assigns a real-value $v(x)$ to a variable $x \in Var$. The set of valuations is given by V. A state of H consists of a location and a valuation $v \in V$. Σ is the set of states. A subset of Σ is called a* region*.*

Labels: *Lab is a finite set of synchronization labels. $\tau \in Lab$ is called the* stutter label*.*

Invariants: *$Inv : Loc \to V$ is a function which assigns each location an invariant.*

Activities: *Act assigns a location $l \in Loc$ a set of* activities*. Each activity is a function $f : \mathbb{R}_{\geq_0} \to V \in Act(l)$. It is required that the activities are time-invariant: for all locations $l \in Loc$, activities $f \in Act(l)$ and $t \in \mathbb{R}_{\geq_0}$, $(f + t) \in Act(l)$ must hold where $(f + t)(t') = f(t + t')$ for all $t' \in \mathbb{R}$.*

Transitions: *$Trans \subseteq Loc \times Lab \times 2^{V \times V} \times Loc$ is a finite set of discrete transitions which are also called* jumps. *A transition is a tuple (l,a,μ,l') where the transition is said to be enabled in a state (l,v) if for some valuation $v' \in V$, $(v,v') \in \mu$ holds. Each location has a* stutter transition *given by $(l,\tau,\{(v,v) \in V^2\},l)$. In literature, a single edge from the transition relation is also called jump. The set of valuations V' enabling a transition is called the* guard *of the transition. The set of valuations defining the valuation after a jump is called the reset of the transition.*

Initial States: *$Init \subseteq \Sigma$ a set of initial states.*

H is called time-deterministic if for every location $l \in Loc$ and valuation $v \in V$, there is at most one activity $f \in Act(l)$ with $f(0) = v$. The activity f is denoted by $\varphi_l[v]$.

In order to further reason about the reachability analysis of hybrid automata, semantics need to be defined. As a hybrid system defines both - discrete and continuous - behavior, a distinction between discrete transitions and time progress in a location is needed. The inference rule[ACH+95] for a discrete jump is defined as

$$\frac{e = (l,a,\mu,l') \in Trans \quad v,v' \in \mu \quad v \in Inv(l) \quad v' \in Inv(l')}{(l,v) \xrightarrow{e} (l',v')}$$

A jump can only be taken if the values of the variables meet certain criteria. The valuations after a reset must satisfy the invariant of the target location, the current valuations must satisfy the guard and v as well as v' must satisfy the reset. The rule [ACH+95] for continuous behavior is more complex.

$$\frac{f \in Act(l) \quad \delta \geq 0 \quad f(0) = v \quad f(\delta) = v' \quad \forall 0 \leq \epsilon \leq \delta . f(\epsilon) \in Inv(l)}{(l,v) \xrightarrow{\delta} (l,v')}$$

Given a time step δ a time transition can be intuitively taken if the continuous behavior in location l does not violate the invariant in l in the interval $[0,\delta]$. The time-can-progress predicate can be used as an abbreviation and simplification of the continuous rule.

$$tcp_l[v](\delta) \quad \text{iff} \quad \forall 0 \leq \epsilon \leq \delta . \varphi_l[v](\epsilon)$$

The inference rule is then

$$\frac{tcp_l[v](\delta) \quad \delta \geq 0}{(l,v) \xrightarrow{\delta} (l,\varphi_l[v](\delta))}$$

We can now define a path of a hybrid automaton.

Definition 2.1.2 (Path [AC15])**.** *A path of a hybrid automaton H is a sequence $\sigma_0 \to \sigma_1 \to ...$ where $\to$ is an execution step, consisting of a discrete step and a time step in H and $\sigma_i \in \Sigma$. A state σ in H is* reachable *iff there is a path of H starting in an initial state σ_0 of H such that $\sigma_0 \to^* \sigma$. $\to^*$ depicts that there is a path starting in σ_0 leading to σ.*

A good example for hybrid automata is given by a bouncing ball model. A ball is dropped from a height y_0. Due to the influence of gravitational force, it accelerates towards the ground with $9.81 \frac{m}{s^2}$. When it reaches the ground, the ball bounces back, is dampened, but is still attracted by gravity. Thus the ball will at some height fall

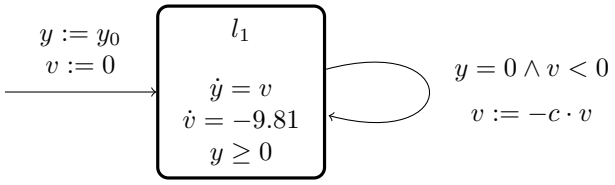


Figure 2.1: Hybrid automaton model of the bouncing ball.

back to the ground. The respective illustration of the hybrid automaton is given in Figure 2.1. A reset is depicted by :=, while guards are directly given as a predicate. Formally the automaton of the bouncing ball is expressed as follows:

Example 2.1.1 (Bouncing Ball [LG09, AC15])**.** *The model of a bouncing ball has two variables - y and v - and only one location: l_1. v describes the velocity of the ball and y is the height of the ball. The dynamic within the location and thus the activities are described by two ordinary differential equations (ODE):*

$$\begin{aligned} Act(l_1) = \{f : \mathbb{R}_{\geq_0} \to V \mid \exists c_y, c_v \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq_0}. \\ f(t)(v) = -9.81t + c_v \wedge \\ f(t)(y) = v \cdot t + c_y\} \end{aligned}$$

The height of the ball is naturally never negative, which can be formulated as an invariant for y:

$$Inv(l_1) = \{val \in V \mid val(y) \geq 0\}$$

As soon as the ball touches the ground its velocity is inverted while losing little of its energy. The last fact is illustrated by the dampening factor $1 > c > 0$. Thus the set of jumps is defined as

$$\begin{aligned} &Trans = \\ &\{(l_1, a, \{(val, val') \in V^2 \mid val(y) = 0 \wedge val(v) < 0 \wedge val'(v) = -c \cdot val(v)\}, l_1), \\ &\ (l_1, \tau, \{(val, val') \in V^2 \mid val = val'\}, l_1)\} \end{aligned}$$

The condition $v < 0$ assures that if $y = 0 \wedge v = 0$ holds, and thus the ball is lying on the ground, the jump can not be taken. If we allowed this condition, we could take the jump infinitely often without letting time progress. This is called Zeno *behavior. Lastly, the initial states are needed. The ball is initially dropped from a height y_0 and has no initial velocity.*

$$Init = \{(l_1, val(y) = y_0 \wedge val(v) = 0)\}$$

where $val \in V$.

Now, as we have introduced hybrid automata as a formal model for hybrid systems, we can advance towards their verification by using reachability analysis.

2.2 Reachability Analysis

Hybrid systems verification is used to reason about the question whether there is a path in a hybrid automaton starting in an initial state leading to a state from a set of

bad states. We consider a system safe when there is no path in the hybrid automaton such that $\sigma_{init} \rightarrow^* \sigma_{bad}$. Thus $R \cap Bad = \emptyset$ holds where R is the set of all reachable states and Bad is a set of bad states. The computation of all reachable states is called reachability analysis. Later on, due to the undecidability of the reachability problem of hybrid automata [HKPV98], we are interested in overapproximated state sets as when $R \subseteq \bar{R}$ holds and $\bar{R}$ is considered safe, then R can also be considered safe. The previously defined semantics of the hybrid automaton state that we can find the set of reachable states by computing two parts - discrete and continuous steps. The first is defined by jumps within the hybrid automaton, the latter by the dynamics of the system within a location. This leads to the definition and computation of the reachable state set of a hybrid system. However, before discussing the algorithmic part of reachability analysis, a formalism is needed.

Definition 2.2.1 (Forward Time Closure [ACH+95]). *Let $l \in Loc$ and $P \subseteq V$. The* forward-time-closure *$FTC(P)_l$ of P in l is the set of valuations that are reachable from some $v \in P$ by letting time progress.*

$$v' \in FTC(P)_l \quad \textit{iff.} \quad \exists v \in P, t \in \mathbb{R}_{\geq 0}.tcp_l[v](t) \wedge v' = \varphi_l[v](t)$$

The reachable states obtained by taking a discrete transition are still undefined. We call this *postcondition* of a set of valuations.

Definition 2.2.2 (Postcondition [ACH+95]). *Let $P \subseteq V$. The valuations reachable from $v \in P$ are gained by executing an enabled transition $e = (l, a, \mu, l')$.*

$$v' \in post_e[P] \quad \textit{iff.} \quad \exists v \in P.(v,v') \in \mu.$$

FTC and $post_e$ can be extended to regions in order to reason about all states within one location and after a jump. A region is denoted by $R_l = (l, P) := \{(l,v) \mid v \in P\}$ where $P \subseteq V$. Then FTC and $post$ [ACH+95] are defined as

$$FTC(R) = \bigcup_{l \in Loc} (l, FTC(R_l)_l)$$

$$post[R] = \bigcup_{\substack{(l,l') \in Loc \times Loc \\ e=(l,a,b,l') \in Trans}} (l', post_e[R_l])$$

where $R = \bigcup_{l \in Loc}(l, R_l)$ and R_l denotes the set of valuations of the region R_l.

A symbolic run of the linear hybrid automaton is a sequence $(l_0, P_0)(l_1,P_1)...(l_i,P_i)...$ of regions such that $P_{i+1} = post_{e_i}[FTC(P_i)_{l_i}]$ where e_i is a transition from l_i to l_{i+1}. This corresponds to a set of runs of H $(l_0, v_0) \rightarrow (l_1,v_1) \rightarrow ...$ where $v_i \in P_i$. Due to the stutter label, the transition relation is reflexive and e_i can always jump from l_i to l_i. This finally gives the notion of reachability. A state is in the reachable region $(I \rightarrow^*)$ of I if there is a symbolic run of H [ACH+95]:

$$\sigma \in (I \rightarrow^*) \quad \text{iff.} \quad \exists \sigma' \in I.\sigma' \rightarrow^* \sigma$$

The backward reachability can be defined similarly. The idea is to start in a bad state σ_{bad} and compute the set of predecessor states $pred = \{\sigma_{pre} \mid \sigma_{pre} \rightarrow^* \sigma_{bad}\}$. When no initial state is an element of $pred$, we consider the system safe. In the course

of the thesis, we concentrate on the forward reachability analysis as a fixed-point search. The class of (linear) hybrid automata is undecidable[HKPV98] in general but the previous definitions yield a semi-decision procedure for the reachability problem in linear hybrid systems. We give the general structure of the fixed-point search in Algorithm 1. The function *ForwardReachability* computes the successor region of R^{new} as a combination of continuous behavior by letting time elapse and computing the jump successor. As $R_i \subseteq R_{i+1}$ holds, we must check whether we already calculated the newly computed set. The algorithm terminates when there are no new sets, and thus a fixed-point has been reached. In the case of the bouncing ball, we see that a fixed-point will not be reached due to the dampening factor. The previous definitions

```
1  Set R^Input; // set of input states
2  Set R;       // set of reachable states
3  R^new = R^Input;
4  R   = ∅;
5  while ( R^new != ∅)
6  {
7      R     = R ∪ R^new;
8      R^new = ForwardReachability(R^new)\R;
9  }
10 return R;
```

Algorithm 1: General Reachability.

yield linear predicates for linear hybrid systems. For example the initial region R_0 of the bouncing ball could be represented by the predicate $(v = 0 \wedge y = 10)$. A full example of the forward reachability using predicates with the previous definitions is given in [ACH+95]. So far we abstracted all set representations. In the following we will describe several ways to represent sets along with their properties.

2.3 State Set Representations

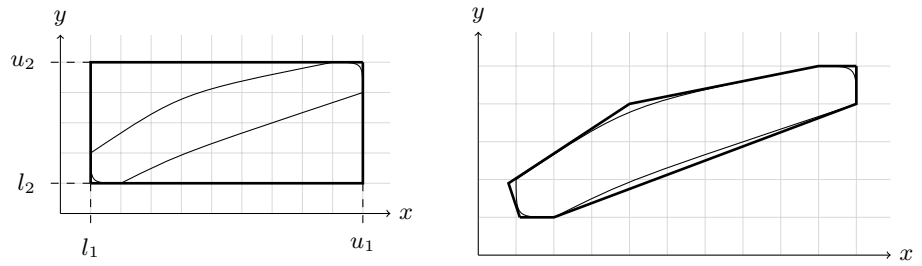
State set representations provide a convenient way to represent and overapproximate the geometric state set of hybrid systems. These representations are suitable as the exactly reachable set can be very hard or even impossible to compute. As previously indicated, we use overapproximation for proving the safety of a hybrid system. Of course, there is the overapproximation $\mathbb{R}^n$ which is capturing the whole state space. However, this overapproximation would always result in insignificant results as it always intersects the set of bad states if they have been defined. If we intersect bad states, we can not conclude unsafety due to the overapproximating character of the computation. Thus one is interested in finding a tight approximation of a state set. As the class of convex sets is easy to represent and efficient methods are known for different operations, we only consider convex sets for the course of this thesis.

Definition 2.3.1 (Box [Moo66, LG09])**.** *A set $\mathcal{B}$ is a box iff it can be expressed as a product of intervals*

$$\mathcal{B} = [l_1, u_1] \times ... \times [l_n, u_n]$$

$\mathcal{B}$ is a set of points x whose i^{th} coordinate lies between the interval bounds, such that $l_i \leq x_i \leq u_i$, where x_i denotes the i^{th} coordinate of a point x.

Boxes are fast and memory efficient but may introduce big overapproximations by default. For example consider an object whose width is much bigger than its height similar to one given in Figure 2.2. Another common geometric representation for a



(a) A box overapproximation of a set. (b) A polytope overapproximation of a set.

Figure 2.2: Representations approximating an object.

set is a convex polytope.

Definition 2.3.2 (Polytope [LG09]). *A polytope $\mathcal{P}$ is the bounded intersection of a finite set $\mathcal{H}$ of halfspaces:*

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

A halfspace is a set defined by a normal vector $n \neq 0$ and a value b:

$$\{x \mid x \cdot n \leq b\}$$

Equivalently a bounded polytope P is the convex hull of a finite set of points $\mathcal{V}$ which is called the set of vertices of $\mathcal{P}$:

$$\mathcal{P} = \left\{ \sum_{v \in \mathcal{V}} \alpha_v v \mid \forall v \in \mathcal{V}, \alpha_v \geq 0 \text{ and } \sum_{v \in V} \alpha_v = 1 \right\}$$

Both definitions lead to complementary representations: a polytope can either be represented by its vertices or by a set of halfspaces. The first is called a $\mathcal{V}$-polytope, the latter is called a $\mathcal{H}$-polytope. As depicted in Figure 2.2 polytopes can have a higher precision than boxes but with every new half space more memory is consumed and more computational time is needed to perform operations on the representation. Support functions are another common way of representing a convex set by a function.

Definition 2.3.3 (Support function [LG09]). *The support function of a set S ρ_S is defined by*

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

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

The support vector $\rho_S(l)$ intuitively represents the *supporting hyperplane* in direction l of S as depicted in Figure 2.3

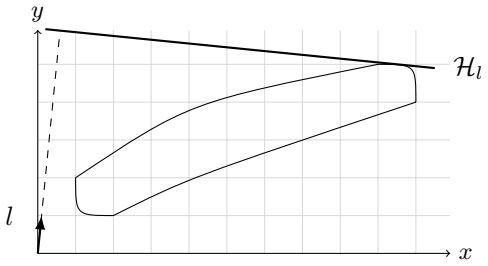


Figure 2.3: Depiction of the supporting hyperplane in direction l [LG09].

2.3.1 Operations

We need several operations on sets for representing behavior like time elapse, guard intersection and resetting variables. The activities in a linear autonomous hybrid system can be expressed as a system of linear ordinary differential equations of the form

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

and solutions have the form

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

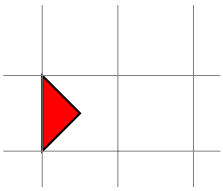
The idea is to compute the time successor $\mathcal{R}_{[0,t]}(\mathcal{I}) = e^{tA}\mathcal{I}$ where $\mathcal{I}$ is a set of initial states. However, we do not know how the state set excluding 0 and t is evolving. Thus we need an overapproximation of the dynamics between 0 and t as the actual state set might not be accurately representable. To obtain an as accurate as possible flowpipe, we discretize the time horizon. The smaller we choose the discretization parameter, the more precise the flow becomes. Now, for a initial set $\mathcal{I}$, we can compute the reachable set in discretized time δ using $e^{\delta A}\mathcal{I}$. This is done iteratively to obtain the reachable states Ω_{i+1} from Ω_i

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

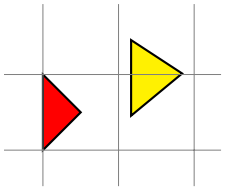
We call a set of reachable states which are obtained by letting time progress a *flowpipe* [LG09]. An example for five flowpipes is given in Figure 2.5. As $e^{\delta A}$ is a matrix defining a linear transformation, we must be able to compute it on a state set representation to let time progress. Before we can compute the flowpipe an initial segment, Ω_0 must be created from the initial state. This is due to the problem that we do not know how the dynamics evolve between time 0 and δ. We give an illustration of the computation of the first segment in Figure 2.4. One computes $\mathcal{R}_{[0,\delta]}(\mathcal{I})$, the time successor of the initial state, and computes the convex hull $CH(\mathcal{I} \cup \mathcal{R}_{[0,\delta]}(\mathcal{I})) = \Omega'$ which captures most of the reachable states but probably not all of the system dynamics. The procedure, which obtains an overapproximation of Ω', that captures the dynamics, is called *bloating*. This requires a Minkowski Sum of a ball of a radius, which depends on the dynamics and the time step. The radius is subject to an optimization problem to guarantee a small error while all dynamics of the system are covered. The set which we obtained from the convex hull is then bloated using this ball. However, there are more sophisticated ways to compute a bloating. We refer the reader to [LG09].

Definition 2.3.4. *Minkowski Sum [Zie12, LG09] The Minkowski Sum of two sets $\mathcal{X}$ and $\mathcal{Y}$ is defined as*

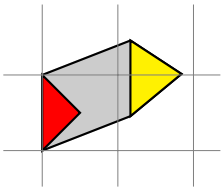
$$\mathcal{X} \oplus \mathcal{Y} = \{x + y \mid x \in \mathcal{X} \text{ and } y \in \mathcal{Y}\}$$



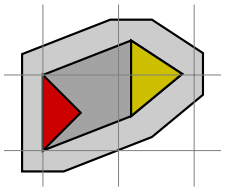
(a) The initial set $\mathcal{I}$.



(b) The initial set $\mathcal{I}$ in red and $e^{\delta A}\mathcal{I}$ in yellow.



(c) The set Ω' obtained by computing the convex hull of the previous segments.



(d) The first segment Ω_0 after bloating the previous set.

Figure 2.4: Creation of the first segment Ω_0.

After covering the dynamics of the current location which is part of the reachability analysis, we may take a jump. We represent by linear constraints, and thus we need to decide the question whether a state set intersects a jump guard. The reset of a jump can be computed using a linear transformation. In general, multiple representations can intersect the same guard and introduce more than one flowpipe. These segments can be aggregated to avoid similar computations. Aggregation is a union of the segments, which intersect the same jump guard.

Thus there are four basic operations which are needed: intersection with halfspaces, convex union, Minkowski Sum and linear transformation Unfortunately, none of the state set representations perform well on all operations as depicted in Table 2.1 and the conversion between them is neither an easy task and therefore we can not have all advantages of all representation types. Table 2.1 shows whether a computation can be done efficiently or not. While $\mathcal{H}$-polytopes perform fast on halfspace intersections, all other operations are hard to compute. On the other hand, $\mathcal{V}$-polytopes perform well in every operation except halfspace intersection. Conversions between the representations need sophisticated algorithms like convex hull and vertex enumeration, and thus one cannot easily use the best of both representations.

2.4 Dynamic Search Strategies

Dynamic search strategies try to recover from a bad state intersection using backtracking to avoid spurious counterexamples. The strategies itself are out of scope for this thesis, but we try to give an intuition to have an understanding of the implementation requirements. Using over-approximating state set representations, we

	$A(\cdot)$	$\cdot \oplus \cdot$	$\cdot \cap \mathcal{H}$	$CH(\cdot \cup \cdot)$
$\mathcal{H}$-polytope	-	-	+	+
$\mathcal{V}$-Polytope	+	+	-	+
Box		+		
Support function	+	+	-	+

Table 2.1: Illustration of the different state set representations and their complexity by operation. $\mathcal{H}$ is a halfspace. Empty fields indicate that the representation is not closed under the respective operation [LG09].

can prove the safety of a hybrid system when a fixed-point is reached, and there is no intersection with a bad state. This is due to the definition of safety. $R \cap B = \emptyset$ implies safety. Then for $\bar{R} \subseteq R$, $\bar{R} \cap B$ is also empty and the system safe. This can not prove unsafety as there might be a computation with a smaller time step and a better or no overapproximation of the representation such that the bad states will never be intersected. At this point a dynamic search which refines the computation and thus reduces the error introduced by overapproximation might lead to a state set which does not intersect the set of bad states anymore. There are two intuitive ways to get a more precise representation. First of all the time step can be reduced further. This leads to a more accurate set, as the smaller time discretization yields smaller flowpipes. The other possibility is to change the representation. For example instead of using boxes as a fast but very rough representation, polytopes may provide a more accurate result. Changing the representation is expensive in general as rather complex algorithms might be needed like vertex enumeration or convex hull. Next, we must choose the set where the backtracking will start to try to recompute. This backtracking point is not intuitive to choose as the system could be indeed unsafe, and the strategies are not able to recover the bad state intersection. The recomputation of the intersecting flowpipe might not be enough as well as the overapproximation of the last flowpipe introduced an error which will always cause the successor flowpipe to intersect the bad states. Thus an implementation must choose a clever backtracking point in order to limit the needed computational steps. Backtracking points can be states on a path to the current state, where branching off paths do not need to be recomputed. Such an example is given in Figure 2.5. The initial segment starts at (0,0) and is linear-transformed along the vertical axis. There are multiple guard intersections which introduce new flowpipes. We will give the example automaton later in Chapter 3 in Figure 3.3.

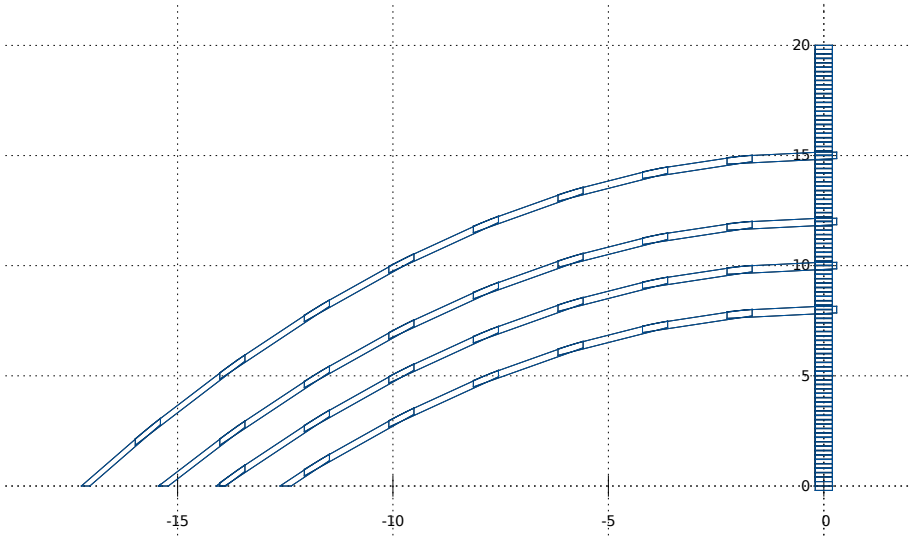


Figure 2.5: One flowpipe starting in (0,0) and introducing four new flowpipes. Elevator example as modeled in Figure 3.3 using $\mathcal{H}$-polytopes. One jump with time step 0.01s and a time horizon of 3s.

Chapter 3

Modularization of Reachability Analysis

The goal of this thesis is to implement a generic approach to hybrid systems reachability analysis. To do so, we need to investigate points of modularization and find a software architecture which supports these points. For this purpose, this chapter discusses general technical requirements to hybrid systems reachability analysis and does a synthesis of modules which are needed for proper analysis. As to the best of our knowledge state of the art, software does not use a parallelized approach for investigating hybrid systems. This chapter introduces a parallelization module and lastly strategies for parallel verification.

3.1 Requirements

The main goal is to provide a modular reachability analysis. For this purpose reconsider the general reachability algorithm from Chapter 2. Given a set of input states this algorithm computes all reachable states until a given condition for termination is reached - like a jump limit or a time horizon - or no new reachable states are found and thus a fixed-point is found. This formulation does not yield parallelization at first sight. When augmenting the previously presented reachability analysis algorithm with concrete datastructures, possible parallelization points become more visible.

```
1  Set<Representation> R^Input; // set of input states
2  Queue<Representation> R;    // set of reachable states
3  Queue<Representation> R^new // not yet investigated states
4
5  R^new.push(R^Input);
6  R = ∅;
7  while ( R^new != ∅ && !termination_condition)
8  {
9      R' = R^new.pop(); // pop gets an element and deletes it from queue
10     R^new.append(ForwardReachability(R')\R); // expensive check
11     R.push(R^new);
12 }
13 return R;
```

This version computes the reachable states incrementally using two queues. R^{new} is responsible for holding all states for that the forward reachability has not been computed, yet. R holds all states that have been explored to check whether a set has already been computed. The last check is expensive regarding computational effort and should be disabled if not needed or heuristics should be implemented. Reconsider the bouncing ball model. The forward reachability method will never return a set which has already been computed as there is a dampening factor c which is reducing the velocity at a jump. Thus the ball will not reach the same height again. To disable the check, an implementation must decide how, where and if the checks are made. However, it is not always possible to determine whether a fixed-point exists. Another advantage of the reformulation is the possibility to compute the forward reachability in parallel. Given an initial state $\mathcal{I}$, we can compute the flowpipe starting in $\mathcal{I}$ without the access to global data structures and therefore in a separate thread. The fixed-point check can be done in the respective thread by reading from R. Thus there are two data structures to be shared - R^{new} and R. These can be changed incrementally without overriding. Additional data can be stored in a thread local storage. The implementation should thus provide multithreading functionality on demand by computing flowpipes in parallel.

An implementation should divide into a layered architecture to force modularity, which leads to better separation of concerns. A library layer should provide abstract classes, thread management, and an execution framework. The execution framework is responsible for executing the forward reachability procedure and thus the flowpipe computation as long as needed. The forward reachability computation, as well as the state set representation, must be interchangeable for the purpose of rapid prototyping. This way new ideas can be tested quickly without making changes in the tool itself. The interchangeability of the state set representation must be implemented as dynamic search strategies might require a different representation. A fast run using boxes as representation might not be enough for proving the safety of a system and a backtracker system decides to redo the computation with a much more precise state set representation or a smaller timestep. Since alongside this thesis dynamic search strategies are implemented, the library layer must provide easy extensibility. As a basis for hybrid systems data structures and reachability analysis, we will use and reuse code from HyPro [SÁC+16]. HyPro is a toolbox for the reachability analysis of hybrid systems providing state set representations. HyPro also implements a reachability algorithm but not in a separate tool.

Another long-term goal is the adaptability of foreign tools. The library part should be sufficiently generic to use other tools for computing forward reachability. As a long-term goal, this could become beneficial for comparing different tools in their runtime by providing a basic measurement layer. For example, this layer may evaluate time consumption of single computations and memory consumption of different tools on equivalent benchmarks.

In a nutshell, the tool to be implemented shall:

1. Provide a reusable library and a command line interface and thus have a clear separation of concerns

2. Provide multi-threading abilities

3. Provide a basic execution framework for hybrid reachability analysis

4. Be easily extensible
5. Be developer and user-friendly for the purpose of rapid prototyping
6. Provide basic measurement methods
7. Reuse existing code from HyPro

3.2 Implementation

In the context of this thesis we implemented a tool. It is called HyDRA which stands for *Hybrid Dynamic Reachability Analysis*. It implements several concepts for a modular hybrid system reachability analysis. It divides into two parts: libHyDRA - or just HyDRA - and HyDRA-cli which is an example usage of libHyDRA using predefined forward reachability implementations taken from HyPro. Thus the implementation reuses much code from HyPro and restructure it into components. We will introduce new components as well if necessary and features which make the development of algorithms for reachability analysis easier.

3.2.1 Worker System

HyDRA defines an abstraction to the forward reachability computation. A worker is the specialization of the *IWorker* interface which is depicted in Figure 3.6 and a *worker instance* is the concrete instantiated object of a Worker. The *IWorker* interface is the entry point to the execution framework provided by HyDRA. It takes care of the correct execution of all worker instances. In order to work correctly, a worker implementation must guarantee reentrancy of the *computeForwardReachability* function and must be stateless. Reentrancy means that a function does not manipulate data structures which are available to multiple threads but only manipulates memory which is passed to the function or is locally available. We need the stateless property for the case that multiple threads are running. One cannot make any assumptions about the order of state set representations which a worker instance may receive. However, HyDRA does not force stateless objects. Doing so could prevent developers from trying out ideas which are making use of the state. Every worker implementation is responsible for the computation of exactly one flowpipe.

At the current state of development, HyDRA makes some assumptions about the data structures a worker may use as currently the dynamic search strategies need a special data structure - *Reachtree Nodes*. These extend a *State* class which contains all necessary information about the used state set representation, the location in the hybrid automaton and the current timestamp. Reachtree Nodes extend this data by several attributes like whether we can use a node as a backtracking point and the kind of representation it holds. The advantage of this approach is that one can easily adapt the used HyPro representations to internal data structures without losing dynamic search strategies. Another method would be using compile time polymorphism for the state set representation - templates in the context of C++. This would result in templating all possible state set representations within the ReachTreeNode class beforehand and affect the ReachTreeNode class which we did not develop in the context of this thesis. If we can forgo dynamic search strategies, we can extend the ReachTreeNode class by a user specific class holding the necessary representations.

3.2.2 Event System

An event system gives the opportunity to react to a set of predefined signals dynamically. This gives the notion of asynchronism by introducing a singleton Eventmanager which can be accessed to dispatch previously registered events. The Manager itself takes care of the correct Eventhandler execution. For more readable code in the particular reachability worker implementation, the Eventhandlers are executed in the context of its calling thread. This way events are not truly asynchronous. In a single threaded environment, the event system is more of a generic operation dispatcher as it gives the opportunity to run a set of methods synchronously using a single entry point.

There is a distinction between user-defined events and system events. System events are predefined within HyDRA and often necessary for internal processing. As shown in Figure 3.1 there is a set of predefined events but for the purpose of rapid prototyping one can easily add custom events which are named by an arbitrary string. This can be useful when synchronization between multiple workers is needed, and there is no predefined event for the respective purpose. Only `ADD_NODE` is

Event name	**Description**	internal	mandatory
`ADD_NODE`	add a new ReachTreeNode	✗	✓
`FINISH_TH`	time horizon reached	✗	✗
`FINISH_FP`	fixpoint reached	✗	✗
`FINISH_BS`	signal: bad states are reached	✗	✗
`PLOT_SEG`	add segment to plotter	✗	✗
`PLOT`	plot all added segments	✗	✗
`HYDRA_STOP_WORKERS`	signal: stop worker threads	✗	✗
`HYDRA_TERMINATE`	signal: hydra shall terminate	✗	✗
`HYDRA_WORKER_IDLE`	signal: a worker is idling	✓	✗

Figure 3.1: List of events used in HyDRA.

mandatory as there is an implementation of this event type that is used internally for adding new nodes to a queue - recall R^{new} in the reachability algorithm. One could still register own implementations of this event in order to attach more logic to node adding. The events which are prefixed with `HYDRA` are currently only used internally for a proper tear down of HyDRA and for solving synchronization problems. It might be beneficial to a user to register own implementations of these events in order to clean up allocated memory or provide log output.

In general, it is not clear when to exit a run. HyDRA uses the fact that there can not be any more work to process when all workers are idling and the queue R^{new} is empty. Thus worker instances register themselves idling when the internal work queue is empty, and they did not receive a new initial state. This gives the responsibility of checking for fixed-points, bad states and a reached time horizon to the worker implementations.

Figure 3.6 shows the how the event system is implemented. The *EventHandler* class is a Singleton object and can be accessed all across the application. In general, all events are specializations of the *IEvent* interface and for convenience there are operations for C++ lambda functions and the support for custom events which are

specific to the respective application. This makes small and events easy to define inline without creating a new class.

3.2.3 Multithreading

We model the reachability analysis algorithm as Producer-Consumer-Problem (PCP) to solve synchronization problems [Hil92]. The producer is the worker instance, and the consumer is a dispatcher, which is responsible for coordinating the worker instances. PCP can be solved efficiently using a buffer data structure. For the specific case of hybrid systems verification, we use a queue as previously indicated. At the beginning of the analysis, the initial states are placed in the queue. The dispatcher dequeues and hands them over to the worker instance. This way the worker instance does not need to take care of selecting the correct element from the buffer and a clear separation of concerns is guaranteed. The worker cannot manipulate the queue directly. Using a dispatcher has the advantage that specialized worker implementations can be used. The dispatcher decides for a given state set representation type which worker instance is suitable. For now, only the state set representation type a worker consumes is important to the dispatcher.

This approach only needs mutual exclusion when enqueueing and dequeueing elements from the queue and is sufficiently efficient. In the following we call the concept of mutual exclusion also locking. In case a worker instance is idle, we can put it to sleep to free more computational power for parallel matrix computations or other running processes. The exact number of CPU cycles to sleep should not be too high as the respective worker instance might be idling although a state set representation is available in the priority queue. The time to sleep should not be too short as well because valuable CPU time will be wasted when no queue element is available. The first version of multithreading in libHyDRA used at least $50\mu s$. We used a random time between $0\mu s$ and $100\mu s$ to wait additionally. This property should ensure that a process which has just woken up has not to wait again to access the queue as the other processes are trying to access the queue, too. However, this can still be inefficient as the queue is repeatedly accessed although it is empty and unnecessary locking happens. We solve the problem by using a system feature - so called condition variables as defined in the C++11 standard. Condition variables are similar to semaphores. They also use signaling to wake up waiting threads but add functionality. Every time an element is enqueued or dequeued in the queue, we must acquire a lock for the reason of thread safety. If no elements are available during a dequeue, we can put the respective thread to sleep until a new element is added or we detect a timeout. To do so, it must first release the lock, must then sleep for some time and wake up when a new element arrives. This technique needs less locking and is thus more efficient.

There are still a few open problems when talking about synchronization in the context of multithreading:

1. Termination of reachability analysis
2. Pausing worker instances
3. Signaling arbitrary behavior (e.g. reaching bad states)
4. Plotting

These problems can be solved best using the Event System introduced in Section 3.2.2. Consider plotting and termination as examples. We should decouple plotting from

the worker. The plotting step is divided into two parts - collecting segments and converting into an output format. Collecting the segments can only be done by the worker but later on, we must call a function, which takes care of the segment processing when the reachability analysis is done. We implement this using two events. First of all the worker instance registers all segments for the actual plot. Next, right after the reachability analysis, the set of segments can be processed by the Plotter. This also has a performance impact on how the plotting should be implemented. For example, if we use support functions, the segments must be preprocessed which takes more time than boxes need. If this is done during the reachability analysis, the runtime will increase dramatically. This multiplies when multiple threads are running as adding a segment to the plotter is an atomic operation. If the preprocessing is done within the event handler, it gets even slower as other threads have to wait for another process to finish. Thus preprocessing and plotting should be done after the reachability analysis so time measurements can be done as precise as possible. The next interesting event takes care of the termination. An empty queue is not a sufficient criterion for termination anymore as a worker instance could be still running although the queue is empty for another thread. To solve this problem every worker instance must register idling when there is no element to process left and working when there is again an element in the queue. Only if all workers are idling the reachability analysis may terminate. Again, we solve this using two events. One for registering the workers state and another for the actual termination indicator.

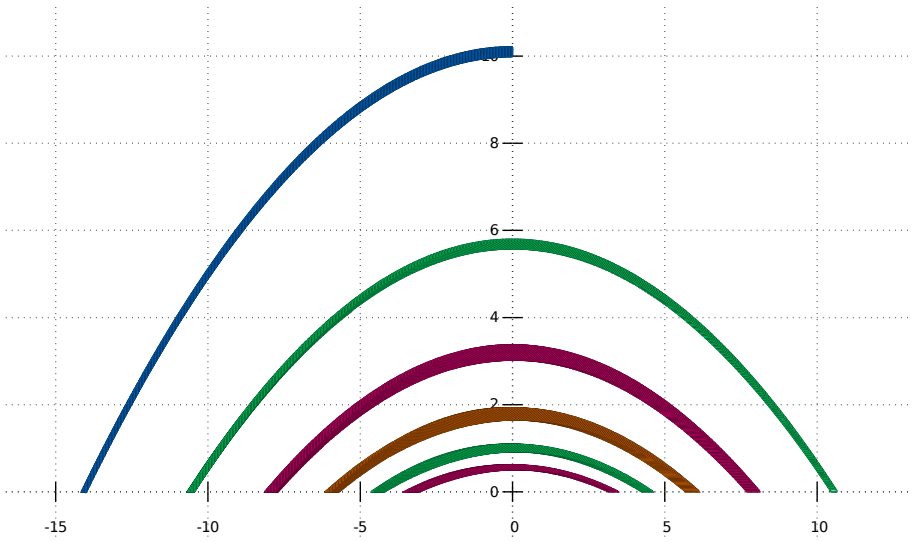


Figure 3.2: Bouncing ball example plot using support functions with the following settings: jump depth: 5, threads: 4, timestep: 0.005s, timehorizon 3s. Each thread is illustrated by a unique color. Velocity is depicted on the horizontal axis, the height of the ball is depicted on the vertical axis.

Figure 3.2 shows a plot of the bouncing ball example reachability analysis run. Four threads were part of the computation. We represent each thread a unique color. Velocity is plotted on the horizontal axis, and the height of the ball is on the vertical

axis. We can see very well that every thread is computing exactly one flowpipe and passes another initial state to the queue using the Eventhandler. It is important to notice that we reuse the different worker instances for another computation. For example, this is illustrated by the two green colored flowpipes. The non-modified bouncing ball example is not suitable for performance improvement measurements when using multiple threads as there is only one jump which we aggregate before passing it to the queue. So there will always be only one thread computing at a time. We extend the bouncing ball example by a lift which is carrying balls up and dropping them from some specified height, to overcome this problem. We give the model in Figure 3.3. This model can be partially computed in parallel. Only the very

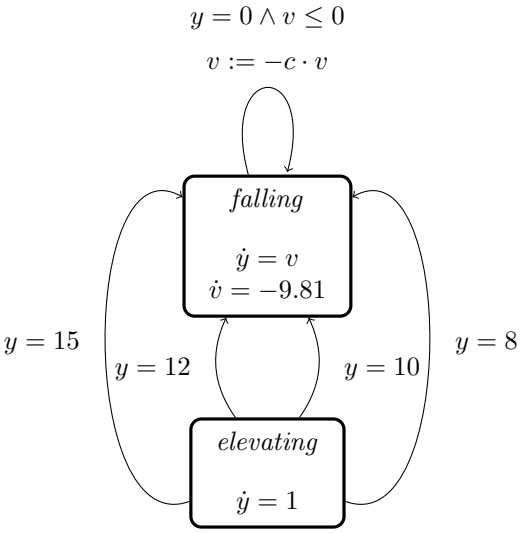


Figure 3.3: Bouncing ball model extended by an elevator which is dropping balls from a specified height. The initial state is $(elevating, v = 0 \wedge y = 0)$.

first flowpipe is computed in a single thread but as there occur four intersections with guards, four new initial states will be added to the queue. A plot for this is given in figure 3.4. The plot shows well which thread computed which flowpipe. This is just a short example for the speedup of the multithreading approach. Further examples are discussed in more detail in Chapter 4.

It remains the question what the best case speedup of a multithreaded reachability analysis run will be. We would expect the best case speedup when we start a computational run with multiple initial states as the same model can be investigated at the same time using multiple threads. If there were no cost of synchronization, we would expect a speedup factor which is equal to the number of processors which we use for the computation. As a reference for all measurements, we make one computational run using only one CPU core. The speedup factor is computed as follows: $\frac{solvetime_1}{solvetime_i}$ where i stands for the number of processors used. Thus one would expect $\frac{solvetime_1}{solvetime_2} = 2$ as when using two CPU cores the computation should be twice as fast. However, in a real-world environment the operating system, the locking techniques, and even memory allocations will reduce this ratio by a value which can hardly

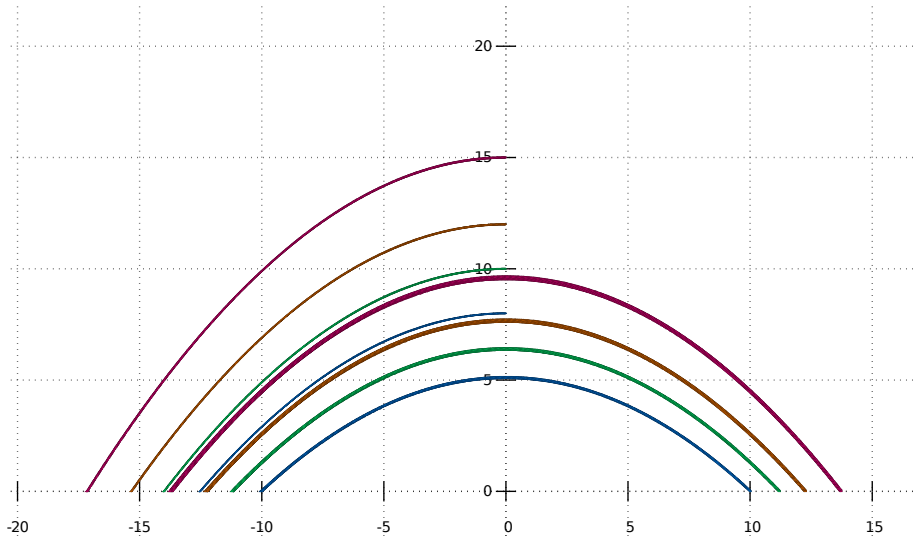


Figure 3.4: Plot of the elevator model using support functions, jump depth 2, a time horizon of 20s, 4 threads and a time step of 0.001s.

number of threads	1	2	3	4
runtime	21.1757s	15.3478s	15.1523s	14.5368s

Figure 3.5: Runtime comparison by thread count of the extended bouncing ball model usind support functions, a jump depth of 6, a time horizon of 20s and a time step of 0.01s.

be estimated. For the elevator example, we already get an idea of the speed up, a parallelized approach can achieve as depicted in Table 3.5.

3.2.4 Library Approach

Our library for reachability analysis aims for easy to use prototype creation, a strict separation of concerns, low overhead and convenience operations for measurements. The strict separation must be done carefully as it has impacts on the usability and the limits of the library. As HyDRA should be usable as a framework and thus not only as a collection of operations, classes and measurement tools for hybrid systems verification, it must provide clear entry points to the framework. We give a minimal application layer example in Chapter A. Three steps are needed to use libHyDRA as an execution framework for hybrid systems verification. First of all, HyDRA must get the worker implementations to know. For this purpose, the *IWorker* interface must be implemented. It forces the existence of the *computeForwardReachability* method which computes the forward time closure of a passed *ReachTreeNode* object. The *ReachTreeNode* class is the state abstraction containing most importantly the state set representation and its metadata. Next, we register workers with the *WorkerProvider*

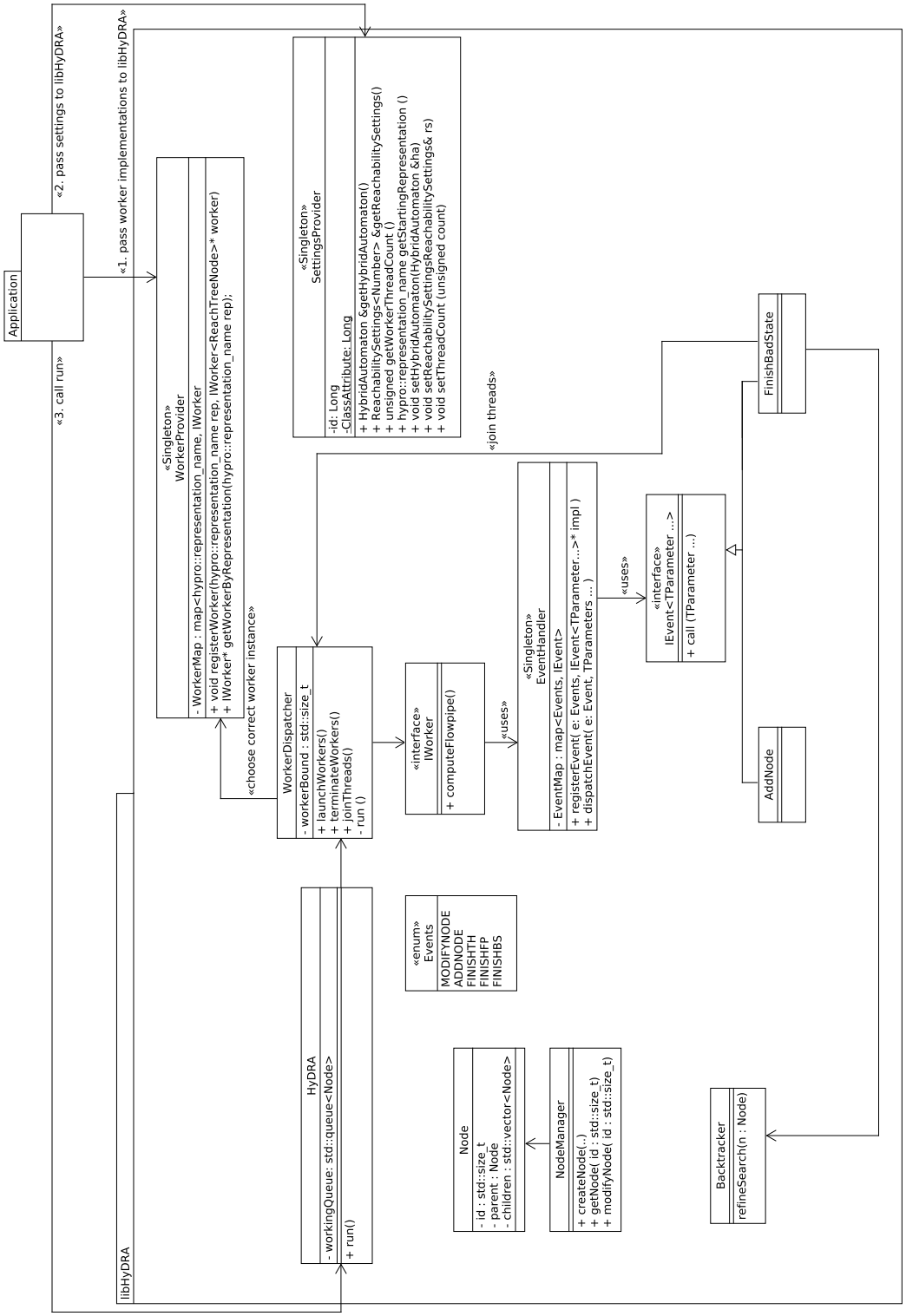


Figure 3.6: HyDRA class diagram

by the type of representation it can handle. After all workers are known to the system, some settings are still missing. First of all, we need the *ReachabilitySettings* which contain information about for example the time horizon or the number of jumps to be taken. Next, we need the *HybridAutomaton* for e.g. the bad states, transitions and dynamics within the system. These are stored in the *HybridAutomaton* data structure. When all parameters are set within the respective singleton classes, we can HyDRA run in a framework like way. For usability purposes, we waive a context object. Using a context object would have the advantage of running multiple HyDRA instances in parallel. This use case should in general not be necessary as it only provides a framework for executing reachability analysis. In case that different models should be checked for safety, the recommended way would be using a shell script instead of parallelizing the analysis within the user application which is using HyDRA.

HyDRA provides multithreading abilities. Thus it is possible to set a number of worker threads to be dispatched. The *WorkerDispatcher* is the central component dispatching threads which then access the work queue and pass *ReachTreeNode*s if existing to the respective worker instance. Every thread is a consumer, executing a producer - the worker instance. The consumer thread does not know the computational result of the executed worker. This is because different kinds of signaling must be done - bad state intersection, adding new states to R^{new} and the Reachtree. By design decision, we decided that it is more in the sense of a framework when the signaling is left to the application developer, as it gives her or him more power. The *EventHandler* singleton class takes implementations of the *IEvent* interface and is executed when *dispatchEvent* is called. The *TerminateEvent* forces HyDRA to terminate. This is for example used by the *WorkerIdleEvent*. It watches the state of all consumer threads.

3.3 Technical challenges

Most difficult when implementing HyDRA was the dependency management and which dependencies might be picked to develop a tool with multithreading abilities. The dependencies must guarantee thread safety or at least reentrancy. HyPro as a library for state set representations was planned for the tool as it is developed at the chair, thus support is fast and much code can be reused and placed into a new tool. However, this means that all dependencies from HyPro have to be reentrant or thread-safe as well. After developing the core structure in HyDRA first tests showed that this is not the case. The test runs even demonstrated that the verification becomes slower when we use multiple threads. As it seemed faster to try out other tools instead of finding the bug in the foreign tool, two other tools (z3 [WHdM09] and Soplex [Wun96]) were introduced to the *Optimizer* class in HyPro as this class is the bottleneck, using about 60 percent of all computational effort. z3 advertises thread safety [WHdM09]. The *Optimizer* is responsible for several tasks like checking a set for consistency and checking for redundant constraints. The setup of z3 is a very time-consuming task, and it seems like z3 is not thread-safe as memory access errors occur in some rare cases. Soplex is a thread-safe simplex solver. After a patch by the Soplex team, the code was thread safe for exact number types, too. This gives the opportunity for future implementations and tests to use different simplex and SMT solvers.

We faced another problem by using the C++ language. For performance and compatibility reasons C++ was chosen as language. C++ provides different concepts for polymorphism. On the one hand inheritance for runtime-polymorphism and on the other hand templates for compile-time polymorphism. HyPro uses templates to abstract the number type. This way a user can choose between several number types - exact and non-exact ones. All information specific to the template instantiation must be known ahead of the compilation. As we planned HyDRd as a dynamic and modular tool which uses multiple state set representations within one run, we had to make assumptions about the template instantiation. One of the assumptions is that mpq_class - an arbitrary precision floating point number type - is always used and we only pass ReachTreeNodes to the Worker processes. The last decision is due to the ease of development and is subject to change. The EventHandler class even uses type erasure to provide a dynamic event system. This technique makes it easier to divide responsibilities and made it possible to develop dynamic search strategies in parallel in another thesis project.

3.4 Strategies for Multithreaded Verification

Parallel computations can lead to great performance improvements if the model introduces more than one flowpipe at one jump or when the verification starts with multiple initial states. Both is currently implemented in HyDRA. Additionally, there are more possibilities to make use of modern hardware and parallelization. An easy way which may lead to better performance are specialized threads which only compute guard and invariant intersections since we need to repeat both operations often. These are basic and rather technical approaches to improve performance. Multithreading can also be used to speed up theoretical concepts.

A naive strategy makes use of the fact that safety can only be guaranteed when there is no intersection with the bad states. When we know that a problem does not scale well across multiple threads, we can check the same problem for safety using another representation in parallel. When one representation can prove safety, we can abort the still running verification task. This is useful when one representation is much more precise than the other one - for example, boxes and $\mathcal{H}$-polytopes. The box representation will be faster but less accurate while the polytopes are much more accurate but slower. When the usage of boxes suffices to show safety, we can abort a parallel verification run which is, for example, using polytopes. The disadvantage of this approach is that we ignore the information provided by the boxes. Every box checks for guard and invariant intersection. Another state set representation might be able to use this information.

First of all, it is necessary to understand that a flowpipe is the result of the time discretization, the guard intersection, and the invariant intersection. It is not essential to the verification run whether the representation is very precise or very rough as we are interested in proving safety as fast as possible. If we were able to skip these intersections by computing them cheaply, it would be possible to skip successor segments and directly compute the time successor of some later point in time of a given state. This is only valid if all points of one representation are contained by the other one to guarantee that the intersection with the more precise representation has already been tested. For this purpose, a bounding box is introduced.

Definition 3.4.1 (Bounding Box)**.** *A bounding box $\mathcal{B}$ of a representation $\mathcal{R}$ is a box such that $\mathcal{R} \subseteq \mathcal{B}$ holds.*

Operations on boxes are known to be fast. The evaluation results confirm this assumption for the tested examples. To formulate a strategy based on the previous idea, we need to prove that the first segment in a box representation is a bounding box of a polytope representation based on a given initial state $\mathcal{I}$ with the same settings. If this holds the guard and invariant intersections can be computed cheaply in the box representation. These intersections are only necessary for a polytope representation when the bounding box intersects the bad states, a guard or an invariant. Thus, if the bounding box does not intersect any of them, we can skip the intersection computations in the polytope representation.

Lemma 3.4.1. *Given an initial state $\mathcal{I}$, which is representable by a finite number of linear constraints, and a time step δ the box representation of the first segment in a hybrid system obtained from $\mathcal{I}$ and δ is a bounding box of the first segment in a polytope representation. Further on the n-th successor segment of the box representation is a bounding box of the polytope segment at time $n \cdot \delta$.*

Proof. Let $\dot{x}(t) = Ax(t)$ be the differential equation describing the dynamics of a given hybrid system H in its initial location, $\mathcal{I}$ be an initial state of H and $\delta \in \mathbb{R}$ an arbitrary but fixed time step. To compute the first segment, we must transform $\mathcal{I}$ into a representation. Polytopes can represent $\mathcal{I}$ exactly as there are only finitely many linear constraints. Boxes will introduce an error as it needs to capture all content of the initial state without being able to use all constraints of $\mathcal{I}$. Thus $\mathcal{P} \subseteq \mathcal{B}$ holds. Now we show that under the assumption that $\mathcal{B} \subseteq \mathcal{P}$ holds, that the operations Minkowski Sum $(\cdot \oplus \cdot)$, linear transformation $(A\cdot)$, convex union $(CH(\cdot \cup \cdot))$ and intersection $(\cdot \cap \cdot)$ are closed under this property. Thus we obtain after application of an operation a representation $\mathcal{P}'$ and $\mathcal{B}'$ such that $\mathcal{P}' \subseteq \mathcal{B}'$ holds. In the following we assume that $\mathcal{P}$ is a $\mathcal{V}$-polytope and we reference an arbitrary but concrete $\mathcal{V}$-polytope by $\mathcal{V}$. As $\mathcal{V}$-polytopes are complementary to $\mathcal{H}$-polytopes, the results are also valid for $\mathcal{H}$-polytopes. We denote that a set is in $\mathcal{V}$-polytope or box representation by $\mathcal{V}(\cdot)$ or $\mathcal{B}(\cdot)$.

$A \cdot$ We apply the linear transformation to $\mathcal{V}$ by applying it to its vertices. This is similar to $\mathcal{B}$ where we first apply the transformation to every point of the intervals of $\mathcal{B}$ and then search for the smallest and the largest point as boxes are not closed under linear transformation. These points define a new box. Thus, this new box $\mathcal{B}'$ is an overapproximation of the linear transformed object. As for every point $x \in \mathcal{V}$, $x \in \mathcal{B}$ holds, we get that if $Ax \in \mathcal{V}'$ then $Ax \in \mathcal{B}'$.

$\cdot \oplus \cdot$ Given two representations A and C represented as boxes or as $\mathcal{V}$-polytopes, the Minkowski Sum for boxes and $\mathcal{V}$-polytopes is computed similarly by calculating the sum of every point or respectively vertex of A and C. As $\mathcal{V}(A) \subseteq \mathcal{B}(A), \mathcal{V}(C) \subseteq \mathcal{B}(C)$ holds, we get for the sum $a+b$ of every point $a \in \mathcal{V}(A)$, $b \in \mathcal{V}(C)$ that $a+b \in \mathcal{V}(A \oplus C)$ and $a+b \in \mathcal{B}(A \oplus C)$. Thus $\mathcal{V}(A \oplus C) \subseteq \mathcal{B}(A \oplus C)$.

$CH(\cdot \cup \cdot)$ Given two representations A and C represented as boxes or as $\mathcal{V}$-polytopes, the convex union is computed by uniting all points of A and C and calculating the convex hull. We get that for every $x \in \mathcal{V}(A)$, $x \in \mathcal{V}(A \cup C)$ holds. As $\mathcal{V}(A) \subseteq \mathcal{B}(A)$ and $\mathcal{V}(C) \subseteq \mathcal{B}(C)$ hold, we get that $\mathcal{V}(A \cup C) \subseteq \mathcal{B}(A \cup C)$.

$\cdot \cap \cdot$ Given two representations A and C represented as boxes or as $\mathcal{V}$-polytopes. $\mathcal{V}'$ denotes $\mathcal{V}(A \cap C)$ and $\mathcal{B}'$ denotes $\mathcal{B}(A \cap C)$. Assume, there is a point $x \in \mathcal{V}'$ such that $x \notin \mathcal{B}'$, then $x \notin \mathcal{B}(A)$ or $x \notin B(C)$ as x must be in A and in C. Then $x \notin \mathcal{V}(A)$ or $x \notin \mathcal{V}(C)$ as $\mathcal{V}(A) \subseteq \mathcal{B}(A)$ and $\mathcal{V}(C) \subseteq \mathcal{B}(C)$ holds. This is a contradiction to $x \in \mathcal{V}'$ and thus $\mathcal{V}' \subseteq \mathcal{B}'$ holds.

Hence, the box representation stays larger than the respective polytope representation. □

This result gives the opportunity to compute a flowpipe using boxes while guaranteeing that every box contains the respective segment when the computation is done using polytopes. Combined with the thoughts from the beginning the guard and invariant intersections can be computed using boxes. A box registers whether there is no intersection at time $n \cdot \delta$ in a globally accessible data structure. In parallel the same initial state is verified using polytopes. Before computing the successor segment, the polytope thread checks whether it is possible to skip the intersection computations. Combined with the naive strategy the verification can be aborted when the boxes prove safety. This strategy could also lead to major improvements in a single threaded environment. For this purpose, every polytope keeps track of its bounding box and first checks whether the bounding box intersects invariant or guard. If not, the polytope thread can skip these steps. This strategy can be advanced easily. When a box thread is much ahead of the respective polytope thread, it is possible to skip the computation of some time successors and directly compute the time successor at $n \cdot \delta$, assuming that the box thread registered that at time $(n-1) \cdot \delta$ no intersections occur using this correlation:

$$\Omega_n = e^{\delta A}\Omega_{n-1} = \underbrace{e^{\delta A} \cdot \ldots \cdot e^{\delta A}\mathcal{I}}_{n-1 \text{ times}} = e^{(n-1)\delta A}\mathcal{I}$$

This is particularly interesting as it might be possible to skip the whole flowpipe computation as safety is guaranteed by the box representation while keeping the precision of the polytopes. When boxes are not able to prove safety anymore, the polytope representation can carry on. Algorithms 2 and 3 illustrate the ideas in algorithms which need to be implemented. The code starting in line 19 is executed by multiple threads. The algorithms also show what is happening during the ForwardReachability function which was introduced in Chapter 2. It is assumed that the algorithms work on global data structures and the names of the functions are chosen to be intuitive for convenience reasons. The algorithms as they are provided are not suitable to be used right away and only provide an idea of how an implementation of the concepts roughly has to look like. The technical details are up to the developer. The algorithm as depicted in Algorithm 3 is easy to adapt for skipping only a few segments and not the whole flowpipe.

```
while ( R^new != ∅ && !termination_condition)
{
    R' = R^new.pop(); // pop deletes from queue
    Flowpipe flowpipe;
    Segment firstSegment = ComputeFirstSegment(R');
    flowpipe.add(firstSegment);
    Segment lastSegment = firstSegment;
    while (ContinueFlowpipeComputation()) {
        if (R'.Type == "polytope") {
            // compute e^{δA} lastSegment
            Segment newSegment = lastSegment.ComputeSuccessorSegment();

            if (!IsIntersectionSkippable(newSegment.TimeStamp)) {
                // must compute intersections
                ComputeIntersections(newSegment);

                if (IntersectBadstates(newSegment)) {
                    // abort as the precise representation intersects bad states
                    abort();
                }
            }
            flowpipe.add(newSegment);
            lastSegment = newSegment;
        } else { // representation is box now
            // compute e^{δA} lastSegment
            Segment newSegment = lastSegment.ComputeSuccessorSegment();

            ComputeIntersections(newSegment);

            if(NoIntersectionOccured()) {
                AddSkippableTimestamp(newSegment.TimeStamp)
            }

            flowpipe.add(newSegment);
            lastSegment = newSegment;
        }
    }
}
return R;
```

Algorithm 2: Algorithm sketch of the first idea. Skip intersection computations of the polytope intersection by computing them cheap using its bounding box.

```
while ( R^new != ∅ && !termination_condition)
{
    R' = R^new.pop(); // pop deletes from queue
    Flowpipe flowpipe;
    Segment firstSegment = ComputeFirstSegment(R');
    flowpipe.add(firstSegment);
    Segment lastSegment = firstSegment;
    while (ContinueFlowpipeComputation()) {
        if (R'.Type == "polytope") {
            // compute e^{nδA} lastSegment - this call may block until
            // a timestamp comes in but only if we are not computing
            // more necessary segments until a guard/inv is hit
            TimeStamp boxIntersectedHere = GetIntersectingTimeStamp();
            Segment newSegment =
                lastSegment.ComputeSuccessorSegment(boxIntersectedHere);

           ComputeIntersections(newSegment);

            if (IntersectBadstates(newSegment)) {
                // abort as the precise representation intersects bad states
                abort();
            }
            flowpipe.add(newSegment);
            lastSegment = newSegment;
        } else { // representation is box now
            // compute e^{δA} lastSegment
            Segment newSegment = lastSegment.ComputeSuccessorSegment();

            ComputeIntersections(newSegment);

            if(IntersectionOccured()) {
                AddTimestamp(newSegment.TimeStamp)
            }

            flowpipe.add(newSegment);
            lastSegment = newSegment;
        }
    }
}
return R;
```

Algorithm 3: Algorithm sketch of the second idea. Skip segment computations of the polytope by computing how far the box gets without intersection.

Chapter 4

Evaluation

The previous chapter already showed a small example for the speed up a multithreaded approach can achieve. For further investigation how the multithreaded approach is performing, we compute the set of reachable states of a given model seven times to eliminate variances from multithreading. Then the best and the worst runtime are erased from the set of measurements to remove outliers introduced by the operating system's scheduler. The result of one benchmark is the arithmetic mean of the remaining five measurements. We show significant runtime improvements in two of the used benchmarks without verifying multiple initial states and reduces the overhead of parallelization to a minimum such that there is no practical need to determine an optimal amount of threads.

4.1 Benchmarks

In a multithreaded environment is particularly interesting to find out the computational overhead introduced by multiple threads - in small and large benchmarks - and the maximum speedup we can achieve by using multiple threads. We measure the first by computing the set of reachable states of a benchmark which introduces exactly one new flowpipe by taking a jump and call this type of benchmark *single-pathed* at it only has a single path. The speedup is calculated in real-world benchmarks like controller models and in artificially enlarged benchmarks by using either multiple transitions - like it has been done previously - or multiple initial states. In the following, we reference different benchmarks according to their model name. Table 4.1 gives an overview on which settings we use.

Bouncing Ball

We investigate four bouncing ball examples. First of all, a small single-pathed bouncing ball example which is fast to compute by using a larger time step and a little jump depth. We compute the set of reachable states in the same benchmark with eight initial states, too. Its purpose is to investigate how big the multithreading overhead is. As the termination of the implementation is only triggered when the set of new states is empty, we expect a big relative cost. To be precise, the worst case is

model	time step	time horizon	jump depth
bouncing_ball_8_init	0.004s	5s	15
bouncing_ball_8_small	0.02s	5s	3
bouncing_ball	0.01s	3s	6
bouncing_ball_small	0.01s	3s	3
cruise_control	0.05s	100s	20
filtered_oscillator	0.01s	4s	15
rod_reactor	0.1s	50s	20
rod_reactor_large	0.01s	50s	30

Table 4.1: Overview of the used benchmark models and the settings used for reachability computation.

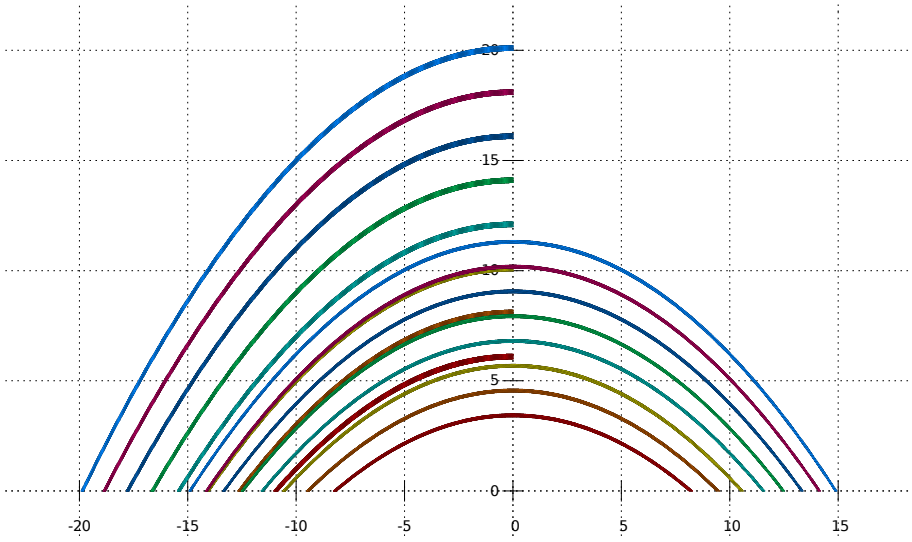


Figure 4.1: Bouncing ball with eight different initial states. $\mathcal{H}$-Polytopes with time step $0.005s$, jump depth 1 and local time horizon $5s$.

that all threads are registering idling and waiting for a new state. Thus the worst case overhead during termination is

$$\mathcal{O}\left(\sum_{T} s_{timeout}\right)$$

where T is the set of worker threads and $s_{timeout}$ is the maximum waiting time for a new state. This is, of course, constant and thus insignificant for long computation times. However, it is the maximum overhead introduced by this kind of implementation of the termination condition. The minimum cost of synchronization we expect is 0. This is the case when the last segment has been computed, the worker returns and looks for a new segment. If all worker threads are idling, the last thread registering

itself idling can initiate termination. The timeout in the queue is set to $100ms$. Thus we expect a maximum overhead of $800ms$ for eight threads. This numbers will differ as this estimation ignores context switching and function calls and asynchronous data structure access. The operating system can also have an impact on the runtime, especially in benchmarks whose set of reachable states is fast to compute. However, by taking multiple measurements, this error is reduced to a minimum. For peak performance improvement measurements, we use the bouncing_ball_8_init model. It uses a small time step, eight initial states and a jump depth of 15. These constraints make the computation last longer and result in a high number of segments and new initial states. We give an example plot in Figure 4.1. The plot shows nicely how the different flowpipes are computed in parallel as each thread is depicted by its unique color.

Rod Reactor

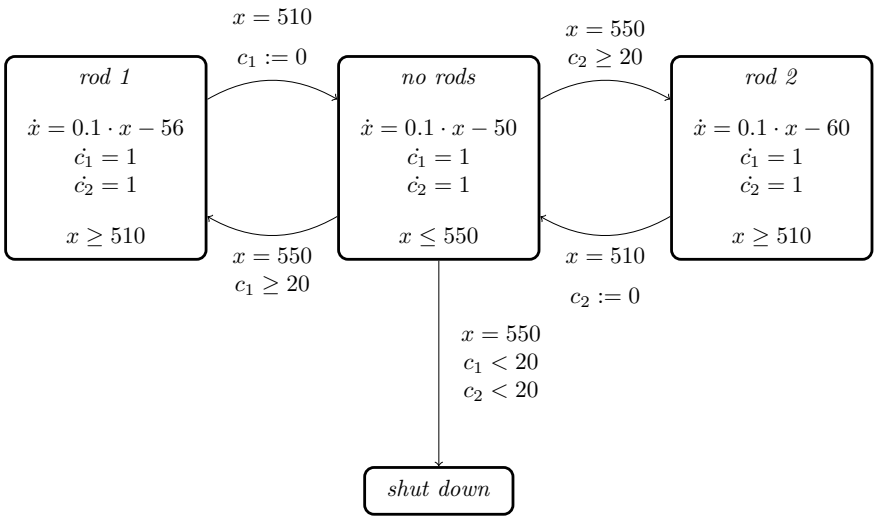


Figure 4.2: Hybrid automaton modeling a rod reactor. x is the temperature variable, c_1 and c_2 are clock variables, measuring the time since the last insertion of rod_i.

The rod reactor is a model of a reactor control system. A reactor consists of a tank and two control rods which differ in their influence on the temperature dynamics of the tank. The water in the tank heats up when no rods are in the water. When the temperature is up to $x_{upper} = 550°C$ and rod_i has not been used for 20 time units it can be inserted into the tank. The tank is cooled down to $x_{lower} = 510°C$ by rod_i and resets its timer to wait again for 20 time units. When the temperature reaches 550°C but there is no rod available the system must be shut down. This is a benchmark which is not deterministic as either rod_1 or rod_2 can be chosen and thus a speed up by using parallelization is expected. Figure 4.3 illustrates well how the system heats up, and is cooled down again by either rod_1 or rod_2.

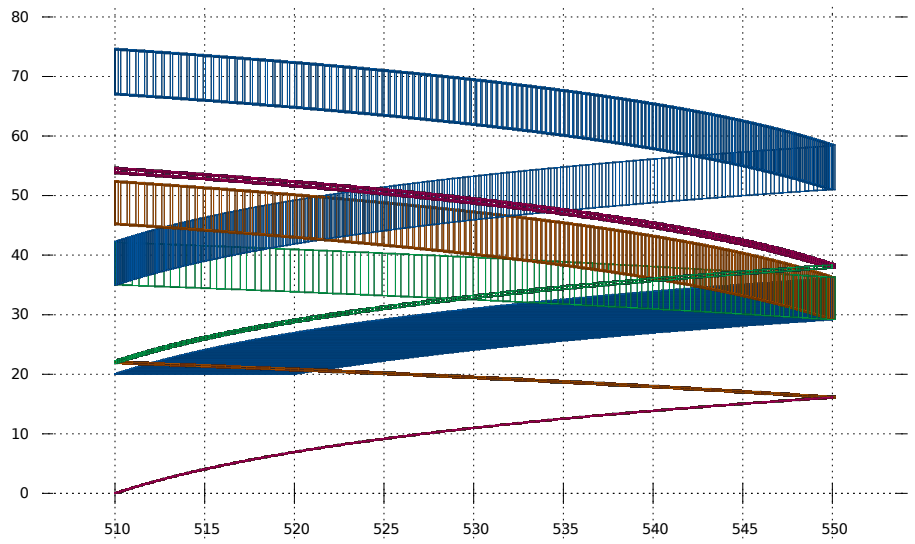


Figure 4.3: Rod reactor example plot using boxes, a time step of $0.1s$, jump depth 20 and local time horizon 50. horizontal axis: temperature, vertical axis: time. The initial state is $(no\ rods, x \in [510{,}520] \wedge c1 = 20 \wedge c2 = 20)$.

Cruise Control

In contrast to the other systems, the cruise control system is a relatively complex hybrid system which models a cruise control. A cruise control is a device which controls the speed of a car. The target speed is set by the driver of the vehicle. To avoid Zeno behavior when the target velocity is reached, an additional state models a waiting time before the car can accelerate or deaccelerate again. For deacceleration, the system is equipped with two kinds of brakes: service and emergency brakes. The service brakes are for small differences between the target and the actual speed while the emergency brakes will only be triggered when a large difference is detected [SÁC+16]. Due to its size, we give the hybrid automaton in Chapter B in Figure B.2.

4.2 Results

The tables in this section show the average runtime and the achieved speedup by the number of threads. Speed up values larger than one mean that multiple threads have been fast in the computation while a value smaller than 1 indicates that more threads have been slower than the reference computation using only one thread. All calculations are done on a single, octa-core Intel Core i7 CPU at 2.8GHz. We do not measure memory consumption as the implementation uses shared memory and thus every thread allocates only memory it needs locally. Every benchmark has about seven gigabytes of RAM available. We give the full measurement results in Chapter B.2.

The results are slightly better than the previously formulated expectations. However, the speed up factor is always near to 1 and the results in Table 4.2 show a positive impact on the performance for all state set representations. The overhead is minimal

i	Box		$\mathcal{H}$-Polytope		Support function	
	⌀	$\frac{t_1}{t_i}$	⌀	$\frac{t_1}{t_i}$	⌀	$\frac{t_1}{t_i}$
1	0.347s	1	2.981s	1	2.589s	1
2	0.342s	1.015	2.938s	1.015	2.491s	1.039
4	0.341s	1.018	2.911s	1.024	**2.469s**	1.049
8	**0.340s**	1.021	**2.905s**	1.026	2.491	1.039

Table 4.2: Results of the bouncing_ball benchmark. i is the number of threads being used. Each row gives the average runtime of all measurements using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speedup factor.

i	Box		$\mathcal{H}$-Polytope		Support function	
	⌀	$\frac{t_1}{t_i}$	⌀	$\frac{t_1}{t_i}$	⌀	$\frac{t_1}{t_i}$
1	21.829s	1	79.975s	1	193.027s	1
2	12.147s	1.797	43.535s	1.837	104.829s	1.841
4	8.302s	2.629	30.423s	2.629	65.613s	2.942
8	**6.000s**	3.638	**22.578s**	3.542	**51.391s**	3.756

Table 4.3: Results of the bouncing_ball_8_init benchmark. i is the number of threads being used. Each row gives the average runtime of all measurements using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speedup factor.

due to the high probability that almost all threads in single-pathed benchmarks are sleeping. The thread which computes the flowpipe to the last initial segment initiates the termination. If all other threads are sleeping, they registered idling before and thus there is no overhead. The results indicate a small speedup of 1 to 4 percent. This speedup is due to the relatively small amount of segments to compute. When multiple threads listen for changes in the queue, a second thread can dequeue an element before the enqueueing worker process returns. However, as soon as the computation time and the number of newly introduced initial states grow, the results get to a factor of close to 1. Thus the impact of the termination condition implementation is even below the expectations. Summarizing, it is possible to run the HyDRA tool with all available threads a system provides without caring about performance loss. Even more impressive is the performance in examples which reveal potential for multithreaded computations. The artificial best case is the reachable state set calculation of a single-pathed system with as many initial states as a system provides CPU cores.

For benchmarking the best case performance improvement we use the bouncing_ball_8_init example. The results are given in Table 4.3. Although they show major improvements in runtime and an average speedup factor of 3.6, using eight threads, the results are behind the expectations. As the results for small examples show practically no overhead for multiple threads, a speedup factor which is nearly equal to the number of used threads would be desirable. Analyzing this problem using the Valgrind [NS07] tool Cachegrind shows that much time seems to be spent in functions provided by GMP. GMP [Gt12] is a C library for fixed-point and arbitrary precision arithmetic. For the bouncing_ball_8_small - the cut-down derivative of the previous benchmark - GMP allocates memory nearly 106 million times and takes 30 percent of the overall time. Memory allocation must be thread safe and locking at a low level is needed as the memory must not be provided to two threads at

i	⌀	$\frac{t_1}{t_i}$
1	1.664s	1
8	**1.007s**	1.652

(a) Cruise Control results (cruise_control).

i	⌀	$\frac{t_1}{t_i}$
1	11.048s	1
2	**6.223s**	1.775
4	7.094s	1.557
8	6.714s	1.646

(b) Rod Reactor results (rod_reactor_large).

Table 4.4: Results of the real-world benchmarks using boxes as representation. i is the number of threads being used. Each row gives the average runtime of all measurements using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speedup factor.

once. This locking could be the cause of the problem, but further investigation is required. Ordinary double numbers should solve the problem. However, we can not work with 64bit floating point numbers easily as numerical issues appear fast, and over-approximation might not be guaranteed anymore. Lastly, we give the results for the real world benchmarks in Table 4.4. They show an improvement of 65 percent in runtime. The rod reactor benchmark illustrates well that the termination condition for multiple threads introduces a small overhead. However, it is roughly within the expected limits. In a nutshell, the HyDRA tool can speed up the computation of common benchmarks by 60 percent and more if it applies to the model. Especially when the verification of multiple initial states is needed, the problem scales well across multiple processors and performance increases by roughly 360 percent.

Chapter 5

Conclusion

The approach of parallelizing the verification of hybrid systems is new and to the best of our knowledge, it has not been done before. To improve the performance on benchmarks which do not scale well on modern computer hardware with multiple cores, it is still possible to make use of other approaches to speed up computations. These methods still require more research in the future. However, parallelization speeds up standard benchmarks and improves the runtime by a magnitude when multiple initial states are subject of the verification task. The developed tool is a good and modular starting point for future researchers who want to try out new ideas quickly. We can extend HyDRA easily by using the event system if needed. Additionally, the library approach provides a simple way to implement so-called workers for reachability analysis. For this purpose, the library provides simple measurement tools, default implementations for the reachability analysis and data structures like a hybrid automaton needed for reachability analysis.

5.1 Summary

Chapter 3 introduces the HyDRA tool which aims to be a modular hybrid systems verifier, building upon much code from HyPro as it ships its own reachability algorithm. After applying minor changes, we integrated it into HyDRA, as well as a parser suitable for Flow* input syntax [CAS13] and some data structures like the hybrid automaton. To make use of modern hardware and provide a forward-looking approach to hybrid systems verification multithreading is part of HyDRA. Along with the producer-consumer problem, we presented three strategies for verification. The simple approach in Chapter 4 shows that the implementation introduces a minimal or even no overhead by multithreading. For common benchmarks, the runtime improves by roughly 60 percent. The results become better when multiple initial states are subject of verification. Then - for eight cores - the improvement rises to up to 360 percent. The reader may review all measurements in the appendix. Although the results for multithreading seem to be promising, they only apply to a small class of current hybrid systems benchmarks. Most benchmarks will not introduce more than one new initial state at once, which makes a naive approach useless. For this purpose, more sophisticated strategies are needed. We presented two promising approaches which may lead to speedups for polytopes as the state set representations. The ideas should be easily extensible to other representations, like for example support functions.

5.2 Future work

There are multiple points of improvements which are still open for discussion.

- HyDRA could become even more generic in its approach by not making any assumptions about the state object. Due to the lack of time and for the ease of development this has not been done, yet. Dynamic search strategies require of course particular information like the time stamp, the settings used for the reachability analysis and the information from the hybrid automaton like invariants, locations, and guards. If we can to abstract this information and provide specific adapters, it would be possible to use HyDRA as a framework for reachability analysis using multithreading while using a custom state implementation and state set representation. The workers are already abstracted. Some similar approach using a state super class throughout the whole code without making assumptions about the kind of representation could make this possible. Currently, there exists a Variant data structure which holds all HyPro implementations for different state set representations. This generalization could make different tools for reachability analysis more comparable as HyDRA already provides basic measurement mechanisms. As a motivation for other tool developers, the multithreading ability should be kept as an essential component of HyDRA.

- There have been problems scaling models across multiple threads. This problem can not be solved completely as one must be able to modify the model such that there are multiple paths. Of course one could waive parallelotope aggregation, but this leads to high runtimes due to an explosion of initial states. Even if there are enough cores available to compute all the new initial states perfectly parallel, the analysis would not be faster than the analysis using aggregation. For this purpose, we introduced three strategies for a faster analysis of these models. Due to the lack of time, we did not implement or test these techniques.

- Another point of investigation regarding the multithreading module is the overhead introduced by GMP and GLPK. Both allocate lots of heap memory which causes the operation system to lock. Double precision numbers would speed up the whole computation but introduce new numerical issues which need to be solved. If these problems are solved a faster analysis is expected. We would also assume that the runtime improvement of multithreading gets closer to the expected factor. Another approach minimizing the memory allocations might be the usage of SoPlex. A comparison of SoPlex and GLPK looks promising as SoPlex advertises itself as one of the fastest open-source simplex solver [ABKW08].

- A fixed-point check is currently not implemented. As a future point for research, this could be done asynchronously by a thread which can abort the computation at any time when a fixed-point is detected. This would decouple this task from the worker threads and maybe it is possible to develop an algorithm which makes use of the information it gets incrementally.

I am curious what HyDRA will become in the future and which ideas will be implemented.

Bibliography

[ABKW08] Tobias Achterberg, Timo Berthold, Thorsten Koch, and Kati Wolter. *Constraint Integer Programming: A New Approach to Integrate CP and MIP*, pages 6–20. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[AC15] Prof. Dr. Erika Ábrahám and Xin Chen. Modeling and analysis of hybrid systems. `https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/vorlesung_hybride_systeme/handout.pdf`, 2015.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. Hybrid systems the algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3 – 34, 1995.

[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*, pages 209–229. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[CAS13] Xin Chen, Erika Abraham, and Sriram Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification (CAV)*, 2013.

[Gt12] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 5.0.5 edition, 2012. `http://gmplib.org/`.

[Hil92] Ralph C. Hilzer, Jr. Synchronization of the producer/consumer problem using semaphores, monitors, and the ada rendezvous. *SIGOPS Oper. Syst. Rev.*, 26(3):31–39, July 1992.

[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.

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

[Moo66] Ramon E Moore. *Interval analysis*, volume 4. 1966.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr 1989.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, June 2007.

[SÁC+16] Stefan Schupp, Erika Ábrahám, Xin Chen, Ibtissem Ben Makhlouf, Sriram Sankaranarayanan, and Stefan Kowalewski. A Toolbox for the Reachability Analysis of Hybrid Systems using Geometric Approximations - HyPro project website. `https://ths.rwth-aachen.de/research/projects/hypro/`, 2016. last access: 30.08.2016.

[VDA98] W. M. P. VAN DER AALST. The application of petri nets to workflow management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

[WHdM09] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo de Moura. *A Concurrent Portfolio Approach to SMT Solving*, pages 715–720. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Wun96] Roland Wunderling. *Paralleler und objektorientierter Simplex-Algorithmus*. PhD thesis, Technische Universität Berlin, 1996. `http://www.zib.de/Publications/abstracts/TR-96-09/`.

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

Appendix A

Examples

Minimal Application Layer Example

```
#include "lib/libHyDRA.h"
#include <string>

int main( int argc, const char** argv )
{
    // start parsing
    hydra::parser::flowstarParser parser;
    hydra::HybridAutomaton ha = parser.parseInput( filename );

    hydra::SettingsProvider::getInstance()
      .setHybridAutomaton(ha);
    hydra::SettingsProvider::getInstance()
      .setReachabilitySettings(parser.mSettings);

    hydra::ReachabilityWorkerProvider::getInstance().registerWorker(
      hypro::representation_name::box,
      new hydra::reachability::ReachabilityWorker
      <hypro::Box<hydra::Number>>(parser.mSettings, ha)
  );
    hydra::ReachabilityWorkerProvider::getInstance().registerWorker(
      hypro::representation_name::polytope_h,
      new hydra::reachability::ReachabilityWorker
      <hypro::HPolytope<hydra::Number>>(parser.mSettings, ha)
  );
    hydra::ReachabilityWorkerProvider::getInstance().registerWorker(
      hypro::representation_name::support_function,
      new hydra::reachability::ReachabilityWorker
      <hypro::SupportFunction<hydra::Number>>(parser.mSettings, ha)
  );

    hydra::run();

    return 0;
}
```

Appendix B

Benchmarks

B.1 Benchmark Automata

All large automata that have been used for benchmarking are listed here as their full appearance in the main part of the thesis would not have been helpful for understanding what is modeled. The model files are available on the HyPro website [SÁC+16].

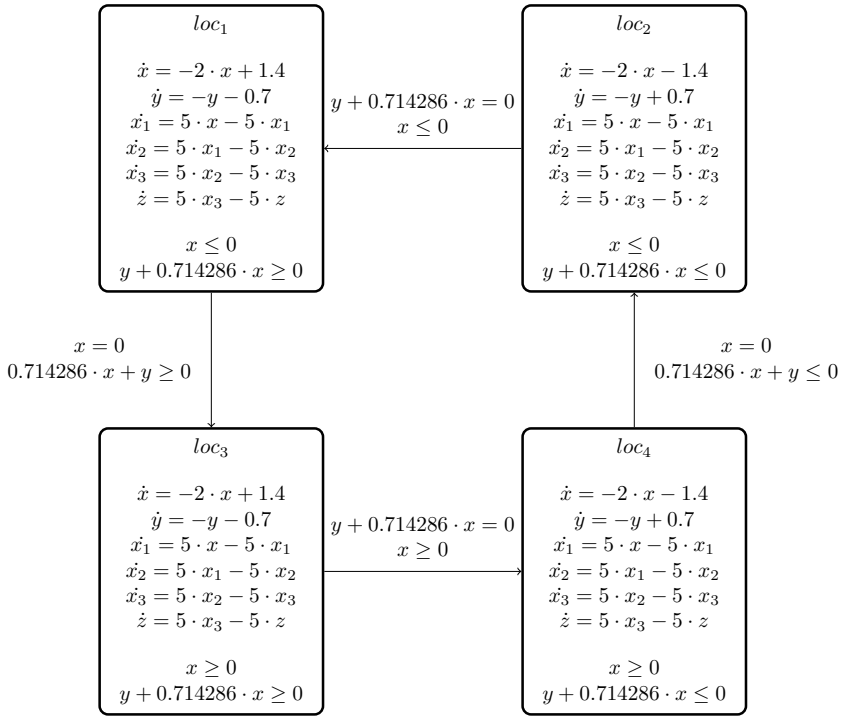


Figure B.1: Hybrid automaton modeling a filtered oscillator [SÁC+16].

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	21.168	21.802	21.243	22.715	21.443	22.250	22.405	21.861	21.829	1
2	12.535	12.846	11.203	14.327	11.508	11.979	11.867	12.324	12.147	1.797
4	8.257	9.258	7.374	9.125	7.496	9.479	7.350	8.334	8.302	2.629
8	6.089	6.080	5.971	5.941	5.904	6.040	5.968	5.999	6.000	3.638

Table B.1: Results for bouncing_ball_8_init benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

B.2 Benchmark Results

The following tables provide the full measurement results as discussed in Chapter 4. The tables were generated from HyDRA log output using a specialized tool written for this purpose in Go. The source code is provided in the HyDRA git repository. Each table shows the following information:

Columns: Iteration n - each benchmark was repeated seven times. The average $\varnothing$ is computed from all values while $\varnothing_{clean}$ was computed from the values after deleting the best and the worst value of all measurements for a representation using a specified amount of threads.

Rows: Threads i - each benchmark was executed using a set of i worker threads. A benchmark was usually executed using either 1, 2, 4 or 8 threads.

All measured values are given in seconds, except the speed up factor of course, which has no unit. The different benchmark settings are defined as

model	time step	time horizon	jump depth
bouncing_ball_8_init	0.004s	5s	15
bouncing_ball_8_small	0.02s	5s	3
bouncing_ball	0.01s	3s	6
bouncing_ball_small	0.01s	3s	3
parallel_example	0.005s	20s	10
cruise_control	0.05s	100s	20
filtered_oscillator	0.01s	4s	15
rod_reactor	0.1s	50s	20
rod_reactor_large	0.01s	50s	30

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	78.150	78.374	78.788	87.873	81.630	80.745	80.339	80.843	79.975	1
2	44.700	44.679	41.477	47.681	41.534	42.977	43.784	43.833	43.535	1.837
4	30.404	32.788	28.494	32.515	27.403	29.183	31.520	30.330	30.423	2.629
8	23.198	22.666	22.835	21.403	22.505	22.723	22.161	22.499	22.578	3.542

Table B.2: Results for bouncing_ball_8_init benchmark with representation polytope_h. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	196.806	195.091	185.390	229.326	185.761	196.078	191.398	197.121	193.027	1
2	106.844	108.719	97.599	183.489	99.391	106.619	102.574	115.034	104.829	1.841
4	67.210	67.600	66.442	65.445	60.562	70.632	61.369	65.609	65.613	2.942
8	53.542	52.388	52.315	49.798	50.194	52.033	50.026	51.471	51.391	3.756

Table B.3: Results for bouncing_ball_8_init benchmark with representation support_function. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	0.476	0.524	0.489	0.461	0.485	0.524	0.469	0.490	0.489	1
2	0.354	0.351	0.347	0.339	0.344	0.353	0.340	0.347	0.347	1.409
4	0.289	0.322	0.313	0.322	0.289	0.323	0.289	0.307	0.307	1.593
8	0.285	0.280	0.283	0.278	0.282	0.282	0.281	0.282	0.282	1.734

Table B.4: Results for bouncing_ball_8_small benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	9.745	10.390	10.251	9.313	9.386	9.733	9.368	9.741	9.697	1
2	5.642	5.838	5.493	5.066	5.304	5.467	5.312	5.446	5.444	1.781
4	3.565	3.545	3.985	3.209	3.352	3.506	3.175	3.477	3.435	2.823
8	2.818	3.015	2.967	2.729	2.805	3.029	2.749	2.873	2.871	3.378

Table B.5: Results for bouncing_ball_8_small benchmark with representation polytope_h. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	7.497	8.966	7.800	7.359	7.267	7.491	7.255	7.662	7.483	1
2	4.028	3.916	3.943	3.642	3.623	4.008	3.691	3.836	3.840	1.949
4	2.533	2.424	2.621	2.312	2.182	2.470	2.531	2.439	2.454	3.049
8	1.947	1.973	2.255	1.998	2.122	2.089	1.992	2.054	2.035	3.677

Table B.6: Results for bouncing_ball_8_small benchmark with representation support_function. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	0.348	0.345	0.344	0.347	0.346	0.357	0.348	0.348	0.347	1
2	0.342	0.343	0.343	0.341	0.341	0.353	0.341	0.343	0.342	1.015
4	0.342	0.339	0.340	0.348	0.341	0.340	0.342	0.342	0.341	1.018
8	0.340	0.342	0.341	0.390	0.340	0.339	0.339	0.347	0.340	1.021

Table B.7: Results for bouncing_ball benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	2.974	2.990	2.987	2.970	2.959	2.999	2.986	2.981	2.981	1
2	2.925	2.955	2.927	2.950	2.942	2.933	2.936	2.938	2.938	1.015
4	2.928	2.898	2.922	3.407	2.892	2.913	2.894	2.979	2.911	1.024
8	2.930	2.885	2.909	3.385	2.886	2.909	2.889	2.970	2.905	1.026

Table B.8: Results for bouncing_ball benchmark with representation polytope_h. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	0.253	0.255	0.253	0.255	0.255	0.255	0.254	0.254	0.254	1
2	0.255	0.252	0.253	0.254	0.254	0.254	0.254	0.254	0.254	1.000
4	0.254	0.254	0.254	0.253	0.254	0.259	0.254	0.255	0.254	1.000
8	0.254	0.261	0.255	0.254	0.254	0.254	0.254	0.255	0.254	1.000

Table B.9: Results for bouncing_ball_small benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	2.195	2.219	2.200	2.226	2.262	2.224	2.201	2.218	2.214	1
2	2.163	2.154	2.163	2.186	2.151	2.172	2.162	2.164	2.163	1.024
4	2.147	2.131	2.129	2.111	2.440	2.138	2.126	2.175	2.134	1.037
8	2.177	2.145	2.199	2.142	2.132	2.138	2.131	2.152	2.147	1.031

Table B.10: Results for bouncing_ball_small benchmark with representation polytope_h. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	1.642	1.654	1.636	1.672	1.650	1.658	1.625	1.648	1.648	1
2	1.599	1.608	1.610	1.688	1.601	1.598	1.600	1.615	1.604	1.027
4	1.599	1.594	1.602	1.589	1.599	1.605	1.617	1.601	1.600	1.030
8	1.606	1.620	1.666	1.605	1.607	1.628	1.599	1.619	1.613	1.022

Table B.11: Results for bouncing_ball_small benchmark with representation support_function. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	2.593	2.557	2.591	2.580	2.586	2.597	2.599	2.586	2.589	1
2	2.484	2.490	2.482	2.500	2.492	2.513	2.491	2.493	2.491	1.039
4	2.480	2.457	2.467	2.643	2.466	2.476	2.443	2.490	2.469	1.049
8	2.498	2.498	2.477	2.535	2.451	2.498	2.486	2.492	2.491	1.039

Table B.12: Results for bouncing_ball benchmark with representation support_function. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	3.586	3.771	3.825	3.579	3.493	3.703	3.551	3.644	3.638	1
2	2.224	2.374	2.325	2.219	2.177	2.346	2.182	2.264	2.259	1.610
4	1.561	1.680	1.543	1.933	1.532	2.090	2.000	1.763	1.743	2.087
8	1.886	1.971	1.846	1.818	1.821	1.875	1.834	1.864	1.852	1.964

Table B.13: Results for parallel_example benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\oslash$	$\oslash_{clean}$	$\frac{t_1}{t_i}$
1	39.929	40.675	39.882	39.778	39.816	39.990	40.486	40.079	40.021	1
2	24.436	24.725	24.382	24.291	24.150	24.219	24.314	24.360	24.328	1.645
4	17.870	18.055	17.736	18.023	17.784	17.645	18.799	17.987	17.894	2.237
8	20.344	19.170	20.374	19.825	20.414	19.636	19.107	19.839	19.870	2.014

Table B.14: Results for parallel_example benchmark with representation polytope_h. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\oslash$	$\oslash_{clean}$	$\frac{t_1}{t_i}$
1	48.278	49.786	47.891	48.017	47.751	49.665	48.449	48.548	48.460	1
2	33.475	33.158	34.452	32.135	32.500	35.212	32.843	33.396	33.286	1.456
4	26.970	27.000	29.659	26.061	26.241	29.157	30.593	27.954	27.805	1.743
8	27.500	28.650	26.939	27.566	27.726	29.310	27.016	27.815	27.692	1.750

Table B.15: Results for parallel_example benchmark with representation support_function. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\oslash$	$\oslash_{clean}$	$\frac{t_1}{t_i}$
1	1.609	1.698	1.731	1.628	1.650	1.693	1.649	1.665	1.664	1
8	1.017	0.725	1.016	1.004	0.996	1.014	1.003	0.968	1.007	1.652

Table B.16: Results for cruise_control benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\oslash$	$\oslash_{clean}$	$\frac{t_1}{t_i}$
1	7.874	7.888	7.869	7.932	7.887	7.910	7.907	7.895	7.893	1
8	7.845	7.888	7.864	7.860	7.860	7.884	7.855	7.865	7.865	1.004

Table B.17: Results for filtered_oscillator benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	1.013	1.021	1.044	1.081	1.028	1.059	1.031	1.040	1.037	1
2	0.654	0.677	0.661	0.679	0.660	0.684	0.655	0.667	0.666	1.557
4	0.659	0.682	0.658	0.930	0.761	0.799	0.653	0.735	0.712	1.456
8	0.702	0.904	0.889	0.737	0.866	0.728	0.828	0.808	0.810	1.280

Table B.18: Results for rod_reactor benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

i \ n	1	2	3	4	5	6	7	$\varnothing$	$\varnothing_{clean}$	$\frac{t_1}{t_i}$
1	10.844	11.325	10.871	10.973	11.169	11.074	11.151	11.058	11.048	1
2	6.353	6.117	6.207	6.936	6.236	6.202	6.060	6.302	6.223	1.775
4	6.757	6.723	7.664	7.722	7.435	6.377	6.893	7.082	7.094	1.557
8	6.905	6.806	6.824	6.571	6.962	6.466	6.334	6.695	6.714	1.646

Table B.19: Results for rod_reactor_large benchmark with representation box. n is the measurement repetition, i is the number of threads being used. Each row gives the runtime of measurement n using i threads. t_i is the average computation time using i threads. $\frac{t_1}{t_i}$ is the speed up factor.

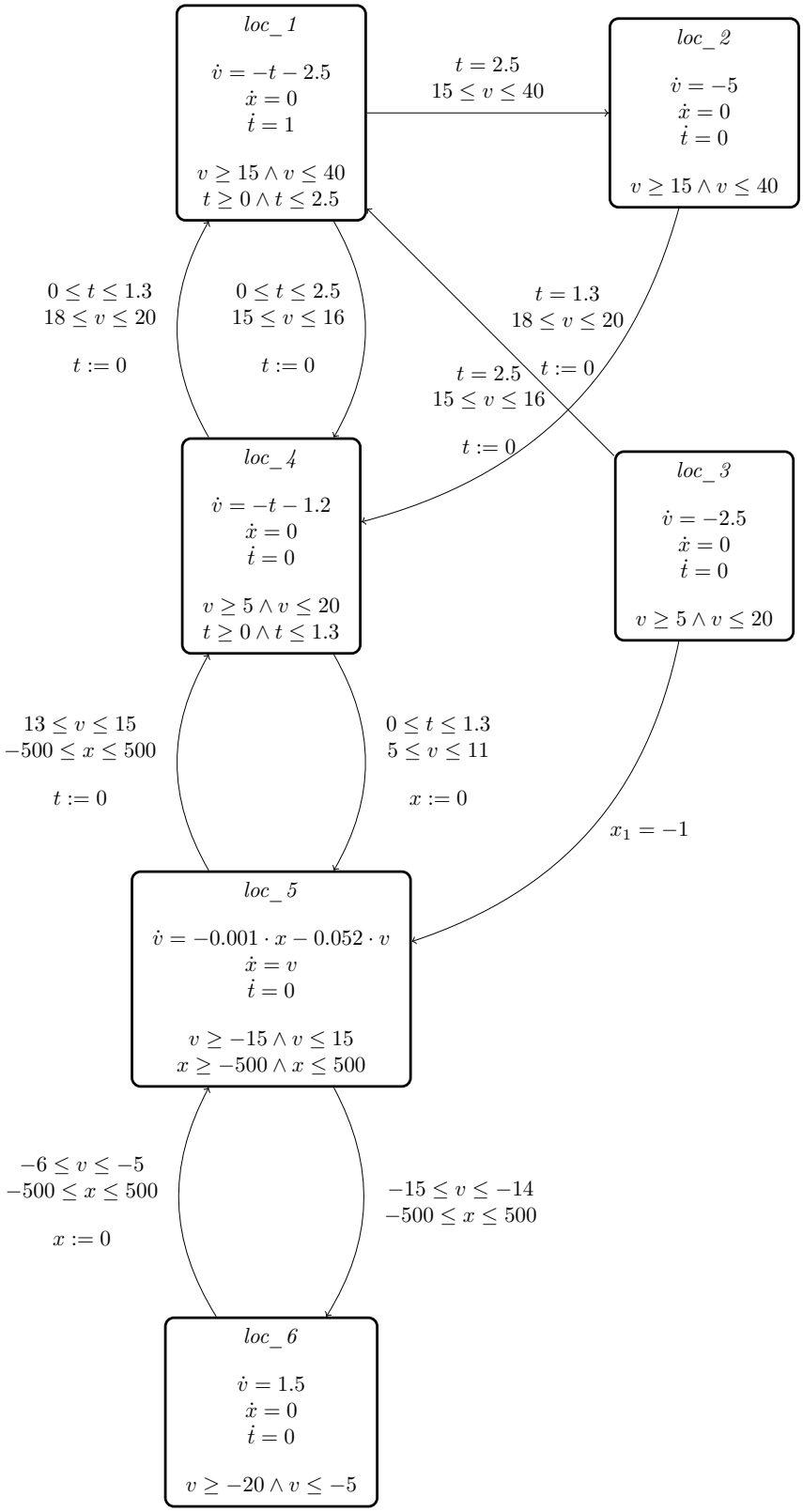


Figure B.2: Hybrid automaton modeling a cruise control [SÁC+16].