MASTER OF SCIENCE THESIS

CEGAR APPROACH FOR HANDLING URGENCY IN HYBRID SYSTEMS

Tristan Ebert

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr. Alexander Mitsos

Additional Advisor:
Dr. Stefan Schupp

Aachen, 11.11.2021

Abstract

Hybrid automata are a popular modeling formalism of hybrid systems, which are systems that have both continuous as well as discrete behavior. To prove safety of hybrid automata, a widely used algorithm is the flowpipe-construction based reachability algorithm which constructs a sequence of convex segments to over-approximate the behavior of the analyzed system in a given time interval. In this thesis we consider the extension of hybrid automata with a set of urgent transitions, which are transitions that enforce discrete change in contrast to the non-deterministic semantics of non-urgent transitions. Applying flowpipe-construction to urgent automata involves the set difference operation which splits the segments into multiple convex sets and can cause an exponential blowup in the number of segments. To mitigate this we apply the CEGAR technique by ignoring the urgency of transitions and refining them on demand with the goal to minimize the number of splits. Experimental results show success of the technique in some instances, provided that urgency would cause a reasonable amount of splitting in the flowpipe segments.

Erklärung

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

Tristan Ebert
Aachen, den 11. November 2021

Contents

Chapter 1

Introduction

There are many examples for hybrid systems we can observe in day-to-day life. These range from simple systems such as a ball bouncing on the floor over complex ones like autonomous cars or even space shuttles to relevant applications such as factory controllers. In contrast to discrete systems, such as a computer program where only instantaneous change of state is present, or continuous systems like the temperature in a room which changes over time, hybrid systems involve both discrete and continuous state changes. This can for example occur when the computer program triggers a heater to be turned on which then affects the temperature in a room.

When analyzing hybrid systems we are often interested whether a given system is *safe*, for example it may be critical to know whether the temperature in a controlled room can exceed an upper limit or whether autonomous cars can crash into one another. To formalize analysis approaches, a common first step is to model hybrid systems as *hybrid automata*, which model the state of a system as a combination of *location* or *mode* and *variable* values. The variables change over time according to a *flow* function while discrete change is modeled by *transitions* between locations. The safety problem can then be formalized by specifiying a set of *bad states* in the automaton which must be unreachable in order for the analyzed system to be safe. Although the safety problem is undecidable [HKPV98], different approaches have been explored in the past to prove safety in some cases for subclasses of hybrid automata.

Here, we focus on *flowpipe-construction based reachability analysis*, which covers the set of reachable states by geometric sets. Thus an over-approximation, i.e., a superset of the set of reachable states is computed, as the exact computation is not possible due to undecidability of the safety problem. In particular, a sequence of *convex segments* is constructed, each of which over-approximates the behavior of the system in a small time interval. By computing enough segments, an arbitrarily large time interval can be covered and if no segment contains a bad state, the system is safe in the time bound.

While this approach can be used to prove safety of some systems, in this thesis we focus on a limitation of hybrid automata as model for hybrid systems, which is that they are *non-deterministic*. In particular, when a transition in an automaton is enabled, the semantics allow both discrete state changes as well as further time evolution. This however does not always accurately reflect the behavior of the considered system, where in some cases discrete change is *urgent* and time is not allowed to elapse further. An example for this is when a heater is turned off as soon as the temperature goes above a critical upper bound. In order to model such behavior more accurately, in this thesis we extend hybrid automata with a set of *urgent transitions* which impede time elapse when they are enabled.

Such an extension has relevant applications since many hybrid systems have naturally urgent semantics. This includes for example the modeling of programmable logic controllers [NÁW15], which are used to control the behavior of plants, or planning problems that occur for example in robotics [BMMW15]. Additionally, other simulation languages for hybrid systems such as SIMULINK or MODELICA use deterministic semantics so that translation to hybrid automata without urgency is difficult [SJ12, MF16a]. Translation is however desirable because simulation is not always sufficient to *prove* safety of hybrid systems.

After formalizing urgency in hybrid automata, the first goal of this thesis is to extend the flowpipe-construction method to handle urgency in order to prove safety for urgent hybrid automata. The main difference here is that we need to exclude states that are only reachable by letting time elapse *after* an urgent transition has already been enabled. Our approach is to use the *set difference* operation in order to exclude the interior of the jump enabling sets of urgent transitions. This however presents a new problem, because we can no longer guarantee convexity of the computed flowpipe segments and therefore *split* them into multiple fragments. For each of these framgnets, successor segments have to be computed which can again be split. Repeating this process can cause an exponential blowup in the number of segments and consequently computation time.

The second goal of this thesis is therefore to apply the Counterexample-guided abstraction refinement (CEGAR) [CGJ$^+$00] technique to the analysis of urgent hybrid automata. CEGAR is a technique with applications to many different areas of model checking, where the idea is to *abstract* the analyzed model in a way that analysis becomes easier while keeping properties of interest. If the abstracted model cannot be verified it is *refined*, which means that features of the original model are added to the abstraction until it cannot be refined further or verification is successful. In our setting we will apply CEGAR by making transitions non-urgent, therefore minimizing the amount of splitting in segments. We will see that this is indeed a sound approach, since by ignoring urgency the computed set of reachable states grow larger, i.e., if the abstracted model is safe then the same holds for the original system. The refinement step in our approach consists of making transitions urgent again, thus making analysis more precise.

Structure. We will start by formally introducing hybrid automata, reachability analysis and in particular flowpipe-construction in Chapter 2, which also requires some results about state set representations. The next step is to extend hybrid automata with urgent transitions, which we do in Chapter 3. In this chapter we also extend flowpipe-construction to urgent automata, highlight problems that can occur and present a specialized algorithm for urgent hybrid automata with constant dynamics. Finally, we are ready to combine the analysis algorithm with the CEGAR technique in Chapter 4 which is the main part of this thesis. Here we explain the general CEGAR technique in more detail, highlight related approaches and formalize a refinement algorithm for urgent hybrid automata. In Chapter 5 we evaluate an implementation of the presented algorithm and compare its performance to that of an analysis algorithm without refinement. We conclude by summarizing our results as well as topics of possible future work in Chapter 6.

Chapter 2

Preliminaries

This chapter introduces preliminary results which will be used in the following chapters. We start by defining hybrid automata as a formal model for hybrid systems and define the safety problem as the main problem of interest in Section 2.1. We then establish basic results about geometric sets and their representations in Section 2.2, where we focus mostly on closed convex polytopes. These results will be used in Section 2.3, where we introduce linear hybrid automata as the main subclass of interest in this thesis and describe the flowpipe-construction based reachability analysis algorithm which will be extended in subsequent chapters.

2.1 Hybrid Automata

In this section, which is mainly based on [Ábr17] and [ACH⁺95, Hen96], we introduce *hybrid automata* and the general *reachability problem.* Hybrid automata aim to model hybrid systems by using a combination of *locations* and *variables* to describe the state of a system. As an example for a hybrid system we consider a moving vehicle which may either be accelerating or braking. This example can be modeled as a hybrid automaton with two locations *accelerating* and *braking* and variables can include the position of the vehicle, given as x coordinate, the velocity and the acceleration. In each location the variables change over time, e.g., in the accelerating location the velocity may increase whereas it decreases in the braking location, and by switching between locations instantaneous change can be modeled. Formally, we define hybrid automata as follows, where we omit components related to parallel composition:

Definition 2.1.1. A *hybrid automaton* H is a tuple

$$H = (\mathit{Loc}, \mathit{Var}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Init}),$$

where the components are defined as follows:

- *Loc* is a finite set of *locations.*
- *Var* is a finite ordered set of real-valued variables $\{x_1, \ldots, x_d\}$. The number of variables d is the *dimension* of H. A *state* of H is a pair of a location and a valuation ν, assigning a real value $\nu(x) \in \mathbb{R}$ to each variable. The set of all states is the *state space* Σ. We denote sets of states that have the same

location component $\ell \in \mathit{Loc}$ as $(\ell, X) = \{(\ell, x) \in \Sigma \mid x \in X\}$. For simplicity we identify valuations with their function values and consider the state space as $\Sigma = \mathit{Loc} \times \mathbb{R}^d$. Similarly, we write x for the vector of variables or the vector of real values, depending on the context.

- *Flow* is a function assigning a set of time-invariant flow functions $\mathit{Flow}(\ell) \subseteq \big(\mathbb{R}_{\geq 0} \to \mathbb{R}^d\big)$ to each location $\ell \in \mathit{Loc}$. Time-invariant means that if $f \in \mathit{Flow}(\ell)$ then $f+t' \in \mathit{Flow}(\ell)$ for all $t' \geq 0$, where $(f+t')(t) = f(t+t')$. H is called *time-deterministic* if for all $x \in \mathbb{R}^d$ there is at most one flow function $f \in \mathit{Flow}(\ell)$ such that $f(0) = x$.
- *Inv* is a function mapping each location to an *invariant* $\mathit{Inv}(\ell) \subseteq \mathbb{R}^d$.
- $\mathit{Edge} \subseteq \mathit{Loc} \times 2^{\mathbb{R}^d \times \mathbb{R}^d} \times \mathit{Loc}$ is a finite set of *transitions*. For $(\ell, \mu, \ell') \in \mathit{Edge}$ we call ℓ the *source location*, ℓ' the *target location* and μ the *jump relation*.
- $\mathit{Init} \subseteq \Sigma$ is the set of *initial states*.

The continuous change of the variables over time is given by the flow functions in each location, where time can elapse as long as the invariant is satisfied, while the discrete changes are defined by the set of transitions. This semantic is formalized by two inference-rules:

Definition 2.1.2. The semantics of a hybrid automaton H are defined by the rules

$$\frac{e = (\ell, \mu, \ell') \in \mathit{Edge} \quad x, x' \in \mathbb{R}^d \quad (x, x') \in \mu \quad x' \in \mathit{Inv}(\ell')}{(\ell, x) \xrightarrow{e} (\ell', x')} \; \mathit{Rule}_{\mathit{discrete}}$$

$$\frac{\begin{array}{c} x, x' \in \mathbb{R}^d \quad \ell \in \mathit{Loc} \quad f \in \mathit{Flow}(\ell) \\ f(0) = x \quad f(t) = x' \quad \forall \varepsilon \in [0, t].f(\varepsilon) \in \mathit{Inv}(\ell) \end{array}}{(\ell, x) \xrightarrow{t} (\ell, x')} \; \mathit{Rule}_{\mathit{time}}$$

Summarizing both rules we define an *execution step*, denoted $(\ell, x) \to (\ell', x')$, as either a time step or a discrete step. An *initial run* is a sequence of execution steps $\sigma_0 \to \sigma_1 \to \ldots$ with $\sigma_0 = (\ell_0, x_0) \in \mathit{Init}$ and $x_0 \in \mathit{Inv}(\ell)$. We also write $\sigma_0 \to^* \sigma_i$ and say that σ_i is *reachable* for all $i \geq 0$.

To define automata, the flow of a location is often given as the solution set to a system of ordinary differential equations (ODE)

$$\dot{x_i}(t) = \varphi_i(x(t)), \text{ for } i = 1, \ldots, d,$$

where $\dot{x_i}$ denotes the derivative of x_i with respect to time and φ_i are quantifier free arithmetic expressions over variable set *Var*. To describe the transitions we often use a combination of *guard* and *reset*. The guard g_e of a transition $e = (\ell, \mu, \ell') \in \mathit{Edge}$ is the set of variable values from which a jump can be taken, i.e., $g_e = \{x \mid \exists x' \in \mathbb{R}^d \mid (x, x') \in \mu\}$. When a transition is taken the variables then change according to the reset function r_e which maps each value to the set of values that can be taken after the transition: $r_e(x) = \{x' \mid (x, x') \in \mu\}$. If $r_e(x)$ is a singleton for all $x \in \mathbb{R}^d$ we also consider the reset as a function $\mathbb{R}^d \to \mathbb{R}^d$. Note that the combination of guard and reset uniquely defines the jump relation μ. We therefore sometimes denote transitions as tuples (ℓ, g_e, r_e, ℓ'). We illustrate the defined concepts on the moving vehicle in Example 2.1.1.

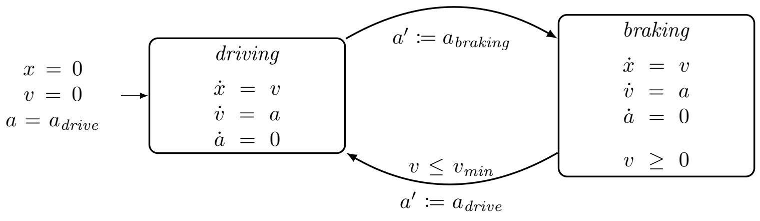


Figure 2.1: Hybrid automaton modeling a moving vehicle.

Example 2.1.1. *We consider the introductory example of a vehicle that is either driving or braking. An automaton modeling the vehicle is depicted in Figure 2.1. Here, we have three variables, x describes the position of the car in one dimension, v is the velocity of the car and a is the acceleration. Initially, x and v are zero and the car is driving. Therefore we start in the location driving with an acceleration $a = a_{drive}$ where $a_{drive} > 0$ is a positive constant. The dynamics are given by the differential equations $\dot{x} = v$ and $\dot{v} = a$, while the acceleration is constant in each location. The car can brake at any time which is modeled by an unguarded transition to the braking location and a reset of a to a_{brake}, which is a negative constant. Here, we assume that the change in acceleration is instant. In the braking location the velocity decreases but can never be negative, which is ensured by the invariant $v \geq 0$ and once some lower threshold $v_{min} \geq 0$ is reached the vehicle can stop braking and return to the driving location.*

A central problem related to hybrid systems is *safety analysis* or the *reachability problem*. Here, we define the set of reachable states

$$\mathrm{Reach}(H) := \{\sigma' \in \Sigma \mid \exists \sigma \in \mathit{Init}.\ \sigma \to^* \sigma'\}. \tag{2.1}$$

Given a set of *bad states* $\mathit{Bad} \subseteq \Sigma$ we want to verify that no bad state can be reached, i.e., that

$$\mathrm{Reach}(H) \cap \mathit{Bad} = \emptyset,$$

in which case we call the system *safe*. In Example 2.1.1 a safety relevant question could be whether the velocity goes above a certain limit v_{max} in which case the set of bad states would be given by $\mathit{Bad} = \{(\ell, (x, v, a)) \in \Sigma \mid \ell \in \{\mathit{driving}, \mathit{braking}\} v \geq v_{max}\}$.

The reachability problem is undecidable for general hybrid automata [HKPV98], however by restricting the possible flow functions, invariants, jump relations and initial sets, subclasses of hybrid automata can be defined. For some of these subclasses the reachability problem is decidable, however in this thesis we are mostly concerned with the subclass of *linear hybrid automata* (see Definition 2.3.1), and we will consider the problem of *bounded reachability analysis*. Here, the runs used to define the set of reachable states in Equation (2.1) are limited by a *time horizon*, which bounds the time that can elapse in a location and a *jump depth*, bounding the total number of discrete transitions. Although even the bounded reachability problem is undecidable for linear hybrid automata we will see in Section 2.3 how in some cases safety can be verified. For that we will employ *flowpipe-construction*-based reachability analysis, which over-approximates the reachable states, i.e., computes a superset, by iteratively transforming geometric sets according to the possible runs of the analyzed automaton.

To define linear hybrid automata and explain the analysis technique in more detail we will first need some results about geometric sets in $\mathbb{R}^d$.

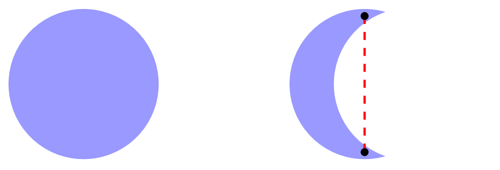


Figure 2.2: Subsets in $\mathbb{R}^2$. The line segment in (b) is not contained in the set.

2.2 State Set Representations

We will now establish some definitions and results about geometric subsets of $\mathbb{R}^d$ and in particular closed convex polytopes, which are central to the following sections. The results of this section can be found with additional details in [Zie95] or [Grü03]. Regarding notation, we consider the points in $\mathbb{R}^d$ as column vectors and denote the i-th component of $v \in \mathbb{R}^d$ by v_i. The transposition into a row vector is written as v^T for $v \in \mathbb{R}^d$. We write $a_{i,j}$ to denote the entry in the i-th row and j-th column of a matrix $A \in \mathbb{R}^{n\times m}$, where $1 \leq i \leq n$ and $1 \leq j \leq m$. Finally, we denote closed intervals in $\mathbb{R}$ as $[a,b] := \{x \in \mathbb{R} \mid a \leq x \leq b\}$, open intervals by $(a,b) := \{x \in \mathbb{R} \mid a < x < b\}$ and denote half-open intervals analogously.

Set properties. We start by defining the set properties *convex, open, closed* and *bounded*. A set is convex if all points on the line segment between two points of the set are also part of the set:

Definition 2.2.1. A set $V \subseteq \mathbb{R}^d$ is called *convex* if for all $u, v \in V$ it holds that

$$\forall \lambda \in [0,1].\lambda u + (1-\lambda)v \in V.$$

Examples for convex and non-convex sets can be seen in Figure 2.2. To measure distance between points we use the Euclidean norm $\|\cdot\|$ and the induced metric $d(u,v) = \|u - v\|$, where

$$\|u\| = \sqrt{\sum_{i=1}^{d} u_i^2}.$$

Using this metric we can define topological set properties. For that we denote the open ball with radius ε centered at a point v by

$$\mathcal{B}_v(\varepsilon) := \{u \in \mathbb{R}^d \mid d(u,v) < \varepsilon\}.$$

Definition 2.2.2. A set $V \subseteq \mathbb{R}^d$ is called *open* if for every point $v \in V$ there exists an $\varepsilon > 0$ such that V contains the ball $\mathcal{B}_v(\varepsilon)$. V is called *closed* if its *complement* $\overline{V} := \mathbb{R}^d \setminus V$ is open. Equivalently, a closed set contains all its *boundary points* where a boundary point of V is a point $v \in \mathbb{R}^d$ such that for all $\varepsilon > 0$

$$\mathcal{B}_v(\varepsilon) \cap V \neq \emptyset.$$

We thus define the *closure* $\mathrm{cl}(V)$ as the union of V and all its boundary points.

Note that both closed and open sets are closed under union and intersection, i.e., the intersection or union of finitely many closed sets is again closed and the same holds for open sets.

We say that $V \subseteq \mathbb{R}^d$ is *bounded* if it is contained in some arbitrarily large ball centered at the origin.

Operations. Flowpipe-construction based reachability analysis heavily relies on specific *operations* on geometric sets, which we will define next. Let U, V be subsets of $\mathbb{R}^d$, let $A \in \mathbb{R}^{d \times d}$ be a matrix and let $b \in \mathbb{R}^d$ be a vector. We define the following operations:

- $U \cap V := \{x \in \mathbb{R}^d \mid x \in V \wedge x \in U\}$ is the *intersection* of U and V,
- $U \cup V := \{x \in \mathbb{R}^d \mid x \in V \vee x \in U\}$ is the *union* of U and V,
- $U \oplus V := \{u + v \mid u \in U \wedge v \in V\}$ is the *Minkowski sum* of U and V,
- $AV + b := \{Av + b \mid v \in V\}$ is an *affine transformation* of V.

Note that convex sets are closed under intersection, Minkowski sum and affine transformations, i.e., if U and V are convex then the result of these operations is convex as well. Since that is not true for the union operation, we define the *convex hull* of a set V which is the smallest convex set containing V:

$$\mathrm{chull}(V) = \left\{ \sum_{i=1}^{n} \lambda_i v_i \,\middle|\, n \geq 1 \wedge \left(\bigwedge_{i=1}^{n} \lambda_i \in [0,1] \wedge v_i \in V \right) \wedge \sum_{i=1}^{n} \lambda_i = 1 \right\}. \tag{2.2}$$

Two Boolean operations that we will make use of are the *emptiness* test, which checks whether a given set is empty, and *containment* test for a point $v \in \mathbb{R}^d$ which checks whether v is contained in a given set.

Representations. In order to use geometric sets algorithmically we require *representations* as well as algorithms for computing the operations introduced above. Here we focus on *polytopes* and in some cases more generally *polyhedra* and discuss some intuitive and useful representations. A more exhaustive collection of representations can be found in [Sch19].

Definition 2.2.3. A *halfspace* h in $\mathbb{R}^d$ is the solution set of a linear inequality

$$h = \{x \in \mathbb{R}^d \mid a^T x \leq z\},$$

where $a \in \mathbb{R}^d$ is the *normal vector* of h and $z \in \mathbb{R}$ is the *offset* of h. Here $a^T x$ is the scalar product of a and x. A *polyhedron* $P \subseteq \mathbb{R}^d$ is the set of points in the intersection of a finite number of halfspaces in $\mathbb{R}^d$. If P is bounded, it is called a *polytope.*

A polyhedron P can always be written as $P = \{x \in \mathbb{R}^d \mid Ax \leq z\}$ where the rows of the matrix $A \in \mathbb{R}^{n \times d}$ are the normal vectors of the defining halfspaces and the vector $z \in \mathbb{R}^d$ contains the corresponding offsets. We call the tuple (A, z) an *$\mathcal{H}$-representation* of P and say that P is given as *$\mathcal{H}$-polyhedron* or *$\mathcal{H}$-polytope* in case P is bounded.

Convex sets can also be represented as the convex hull of a set of points and in case this set of points is finite it can be shown that the resulting set is a polytope [Zie95]. Conversely, for every polytope P there is a finite set of points $V(P)$, called *vertices* of P such that $\mathrm{chull}(V(P)) = P$. Note that this does not hold for unbounded polyhedra or general convex sets which can have an *infinite* set of vertices. We thus call $V(P)$ a *vertex-* or *$\mathcal{V}$-representation* of P and say that P is a *$\mathcal{V}$-polytope.* While conversion between $\mathcal{V}$ and $\mathcal{H}$ representations is possible, no polynomial algorithms are known.

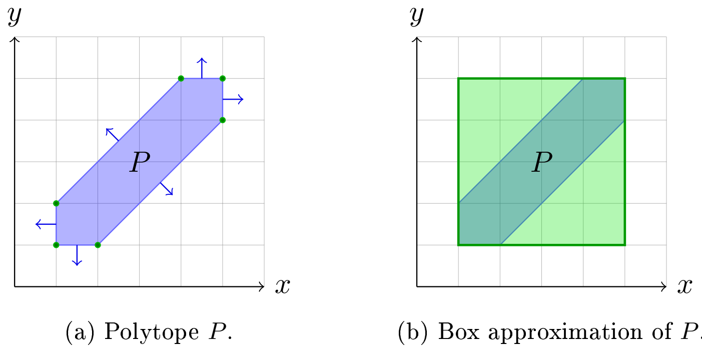


Figure 2.3: A polytope with its vertices (green points) and the normal vectors of its defining halfspaces (blue arrows) in (a) and the bounding box (green) in (b).

Figure 2.3a shows an example of a polytope with the normal vectors of its defining halfspaces and its vertices.

Finally, we consider boxes as a computationally cheap but less precise set representation.

Definition 2.2.4. A *box* $\mathcal{B}$ of dimension d is a subset of $\mathbb{R}^d$ that can be written as the Cartesian product of d intervals $I_1, \dots, I_d \subseteq \mathbb{R}$.

$$\mathcal{B} = I_1 \times \cdots \times I_d.$$

A box can be uniquely represented by the list of tuples

$$((I_1^\ell, I_1^u), \dots, (I_d^\ell, I_d^u)),$$

where $I_j = [I_j^\ell, I_j^u]$. Boxes cannot be used to exactly represent polytopes in general, since not all polytopes are box shaped and computation with boxes is therefore usually less precise than with polytopes. An example for a polytope that is not box shaped and its over-approximating bounding box is shown in Figure 2.3b.

The choice of set representation plays an important role in reachability analysis because the computational complexity of the set operations can vary greatly with the used representation. Additionally, different representations can imply a tradeoff between speed and precision, e.g., operations on boxes are usually fast but less precise. In the following we assume that the set operations defined above are available for boxes and polytopal representations and refer to [Sch19] for specific algorithms and an analysis of computational complexity. If the result of a computation cannot be represented exactly by the input representation, like for example the union of two $\mathcal{H}$- or $\mathcal{V}$-polytopes, we assume that the result is over-approximative.

2.3 Reachability Analysis

We now return to the reachability problem, where we want to know for a given hybrid automaton H and a set of bad states $Bad \subseteq \Sigma$ whether $\text{Reach}(H) \cap Bad = \emptyset$. While the problem is in general undecidable, various approaches have been developed to partially solve the problem e.g., for subclasses of hybrid automata. Here, we consider the *bounded reachability problem* with a time bound and a jump depth bound for time deterministic *linear hybrid automata* (LHA):

Definition 2.3.1. A hybrid automaton $H = (Loc, Var, Flow, Inv, Edge, Init)$ is *linear* if the flow in each location is given as the solution set to a system of linear ODEs

$$\dot{x} = Ax,$$

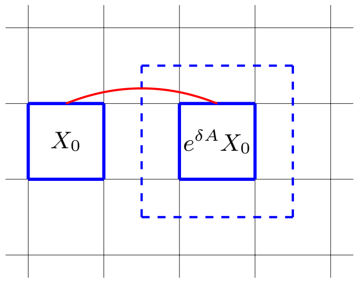
for a matrix $A \in \mathbb{R}^{d \times d}$, all invariants, guards and initial sets are closed polytopes and all resets are affine transformations $r(x) = Rx + c$ where $R \in \mathbb{R}^{d \times d}$ and $c \in \mathbb{R}^d$.

Note that also flows of the form $\dot{x} = Ax + b$ can be encoded in LHA by adding an additional constant variable. In particular we allow the flows for variables to be constant and if all flows in all locations are constant we say that H is an LHA I. The more general class we defined as LHA is sometimes referred to as LHA II, but since we will almost always work with this class we simply call them LHA for brevity. In addition to LHA we assume that the set of bad states is a polyhedron in each location, i.e., $Bad = \cup_{\ell \in Loc}(\ell, Bad_\ell)$, where Bad_ℓ are potentially empty polyhedra.

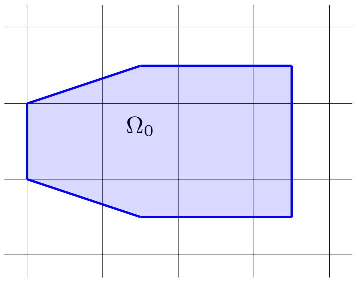
We will now describe the ***flowpipe-construction based reachability analysis*** algorithm [CK98, Gue09], which is summarized in Algorithm 1. Here, we follow the presentation in [Ábr17] and [Sch19]. The idea is to compute an over-approximative set of states $\text{Reach}'(H)$ that contains all actually reachable states $\text{Reach}(H)$. Safety can then be verified by checking that $\text{Reach}'(H) \cap Bad = \emptyset$. Note that if $\text{Reach}'(H) \cap Bad \neq \emptyset$ then we don't know whether the system is safe or unsafe since the bad states could lie only in $\text{Reach}'(H) \setminus \text{Reach}(H)$. Algorithm 1 starts with the set of initial states *Init* and alternatingly computes the states that are reachable by letting time elapse or by taking a discrete transition. This is repeated until either no new jump successors are found or until the jump depth is reached, at which point all computed states are returned. Time successors, up to a time bound, are computed with the *computeFlowPipe* function and jump successors with *computeJumpSuccessors*. We will next explain how these functions work in more detail.

Algorithm 1: Forward reachability analysis

```
Input  : A linear hybrid automaton $H$ with initial set $Init$.
Output: Over-approximation of $\text{Reach}(H)$.
1  $R := Init$;
2  $R_{new} := \{Init\}$;
3  while $R_{new} \neq \emptyset$ do
4      Let $stateset \in R_{new}$;
5      $R_{new} := R_{new} \setminus \{stateset\}$;
6      $R' := computeFlowPipe(stateset)$;
7      if $!jumpDepthReached()$ then
8          $R_{new} := R_{new} \cup computeJumpSuccessors(R')$;
9      end
10     $R := R \cup R'$;
11 end
12 return $R$
```



(a) The red trajectory is not contained in the convex hull of X_0 and $e^{\delta A}X_0$.



(b) The first segment after bloating.

Figure 2.4: Bloating is used to include non-linear trajectories in the first segment.

Flowpipe-construction. We fix an initial state set (ℓ, X_0) where X_0 is a closed polytope. The goal is to compute an over-approximation of the time successors within a given time horizon T.

The flowpipe algorithm discretizes the time interval $[0, T]$ into $N \in \mathbb{N}_{>0}$ time steps of equal size $[0, \delta], [\delta, 2\delta], \ldots, [(N-1) \cdot \delta, N \cdot \delta]$ where $\delta = \frac{T}{N}$. We then compute the *segments* Ω_i such that each Ω_i covers the states reachable in the time interval $[i \cdot \delta, (i+1) \cdot \delta]$. The union of the segments then gives an over-approximation of the time successors of the initial states.

To compute the segments, recall that the flow of a location in an LHA is given by an ODE of the form $\dot{x}(t) = Ax(t)$. A solution to this ODE is given by $x(t) = e^{tA}x(0)$. Therefore the segments Ω_i can be computed with the recurrence relation

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

which can be approximated using the formula

$$e^{\delta A} = \sum_{n=0}^{\infty} \frac{A^n}{n!}.$$

To compute the first segment we need to over-approximate the states reachable from X_0 in the time interval $[0, \delta]$. A first approach is to take the convex hull of X_0 and the states reachable at time δ, given by $X_\delta = e^{A\delta}X_0$. This however does not include the non-linear trajectories described by the ODE as can be seen in Figure 2.4. Therefore we use *non-uniform bloating* which means that X_δ enlarged using a ball $\mathcal{B}$ where the radius depends on δ, A and X_0:

$$\Omega_0 = \text{chull}(X_0 \cup (X_\delta \oplus \mathcal{B})).$$

For a detailed description of the bloating technique we refer to [Gir05] and for details on non-uniform bloating to [Gue09].

Finally, to take the invariant $Inv(\ell)$ into account, each segment is intersected with the polyhedron induced by the invariant. Note that if for some i there is an empty intersection $\Omega_i \cap Inv(\ell) = \emptyset$, we don't need to compute subsequent Ω_j for $j > i$ because the states are no longer reachable.

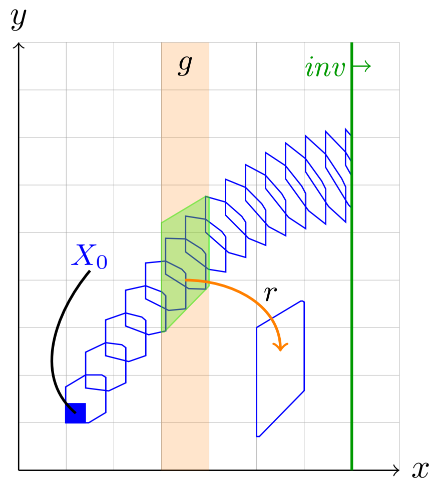
Jump successors computation. Given a set of segments $\Omega_0, \ldots, \Omega_{N-1}$ in a location ℓ we want to compute the states that are reachable by executing a discrete jump. Multiple outgoing transitions from ℓ are treated independently so we explain how to deal with a single transition $e = (\ell, g, r, \ell')$.

The basic approach is to intersect each segment with the polyhedron generated by the guard g of e, which results in a family of polyhedra, called *jump predecessors*. The reset, defined by an affine transformation, can then be applied to each of the jump predecessors, giving the jump successors in ℓ'.

Treating each segment individually can however lead to an exponential blowup in computation time, since for each jump successor the time successors have to be computed again. This can be improved by using *aggregation* or *clustering*. Aggregation means that all jump predecessors are treated simultaneously by applying the reset to the convex hull of the predecessors. The advantage is that we only get one jump successor, however aggregation can also lead to additional over-approximation. Clustering on the other hand is a middle ground approach where the number of jump successors is limited by an upper bound c. The jump predecessors are divided into at most c families of sets which are then aggregated respectively.

The flowpipe-construction algorithm with aggregation is illustrated in Figure 2.5, where the flowpipe segments for one time elapse and the jump successors for a discrete jump are computed.

Figure 2.5: Flowpipe-construction with aggregation from initial set X_0.



Reachtrees and path analysis. The alternating computation of continuous and discrete successors motivates a natural datastructure to store the computed reachable sets and the corresponding paths. A *reachtree* of a hybrid automaton H is a graph whose nodes represent states that are reachable by letting time elapse in a location from an initial set and whose edges represent discrete transitions. Reachtrees are useful whenever we want to backtrack in the computation, e.g., to find potentially unsafe paths. We define reachtrees similarly to [Sch19].

Definition 2.3.2. For a hybrid automaton

$$H = (\mathit{Loc}, \mathit{Var}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Init}),$$

we define a *reachtree* as a tuple

$$T_H = (\mathit{Nodes}, \mathit{Root}, \mathit{Succ}, \mathit{State}, \mathit{Trace}),$$

where

- *Nodes* is a finite set of nodes,
- $\mathit{Root} \in \mathit{Nodes}$ is the root of the tree,

- $\mathit{Succ} \subseteq \mathit{Nodes} \times \mathit{Nodes}$ is a set of edges, such that $(\mathit{Nodes}, \mathit{Root}, \mathit{Succ})$ is a tree, i.e., contains no circles,
- $\mathit{State}\colon \mathit{Nodes} \to \mathit{Loc} \times 2^{\mathbb{R}^d}$ assigns each node an initial set,
- $\mathit{Trace}\colon \mathit{Succ} \to \mathit{Edge} \times \mathbb{I}_{\mathbb{R}}$ assigns each edge in T_H an edge in H and a real interval $I \in \mathbb{I}_{\mathbb{R}}$, indicating the time of the jump.

Note that the nodes of the tree only store the initial state set and not all time successors, which can differ from actual implementation where it may be useful to store the computed flowpipes as well. In the theoretical treatment we assume that flowpipes can be computed on demand: For a node N in a reachtree, we define the operator $\mathrm{FP}(N)$, which is the set of segments computed by the flowpipe-construction algorithm. Similarly, we write $\mathrm{JS}(N)$ to denote the set of jump successors of N, i.e., the state sets resulting from applying guard intersections and resets to the segments in $\mathrm{FP}(N)$. Note that both $\mathrm{FP}(N)$ and $\mathrm{JS}(N)$ depend on the used analysis parameters such as time step size or whether aggregation or clustering is used. In the following we assume that these parameters are fixed beforehand and never modify them, so that the operators are well defined.

Given a reachtree T_H we call a node $N \in \mathit{Nodes}$ *complete* in T_H if N has a corresponding child for every jump successor i.e., if

$$\mathrm{JS}(N) = \{\mathit{State}(N') \mid (N, N') \in \mathit{Succ}\}.$$

Similarly, a reachtree T_H is called *complete of depth J* if every node of T_H with depth at most J is complete.

Instead of computing the reachable states on every path, it will also be useful to only analyze specific *analysis paths:*

Definition 2.3.3. An *analysis path* is a tuple $p = (\tau_1, e_1, \tau_2, e_2, \dots, \tau_n, e_n)$ where each τ_i is a real interval and $e_i \in \mathit{Edge}$. We write $|p| = n$ for the length of the path and denote the unique path of length 0 with $\emptyset$.

The intervals τ_i are time intervals in which the transition e_i can be taken. For an analysis path p we therefore define the set of nodes nodes(p) belonging to p inductively:

- If $|p| = 0$ then $\mathrm{nodes}(p) := \{\mathit{Root}\}$.
- If $p = (\tau_1, e_1, \dots, \tau_n, e_n)$ for $n \geq 1$ then

$$\begin{aligned}\mathrm{nodes}(p) := \mathrm{nodes}(p') \cup \{N' \mid{} & \exists N \in \mathrm{nodes}(p').\ \mathrm{depth}(N) = n-1 \wedge \\ & (N, N') \in \mathit{Succ} \wedge \\ & \mathit{Trace}(N, N') = (\tau, e) \wedge \tau \cap \tau_n \neq \emptyset \wedge e = e_n\},\end{aligned}$$

  where $p' = (\tau_1, e_1, \dots, \tau_{n-1}, e_{n-1})$.

In other words, we iteratively add all nodes that can be reached by taking the next transition on the path in the given time interval.

We call p *complete* in T_H, if every node $N \in \mathrm{nodes}(p)$ is complete in T_H. Thus, if an analysis path is complete in T_H then the flowpipes of the nodes in T_H contain all states that are reachable on any run in H that is induced by p. We denote the corresponding state set by $\text{Reach}_{T_H}(p)$, which is defined by the flowpipes of the nodes belonging to p:

$$\text{Reach}_{T_H}(p) = \bigcup_{\substack{N \in \text{nodes}(p) \\ S \in \text{FP}(N)}} S.$$

If T_H is clear from the context, we may omit it and write $\text{Reach}(p)$ instead.

Note that we explicitly differentiate between runs of an automaton and analysis paths although both are related: An analysis path induces arbitrarily many runs, while each run induces a unique analysis path. In Chapter 4 it will however turn out beneficial to tie analysis paths to the explored reachtree rather than the automaton because the latter may change during refinement.

Given an analysis path p, Algorithm 2 creates a reachtree in which p is complete and returns $\text{Reach}(p)$ as the result. Here, we use the node-operator that creates a node with the given initial state set and parent node. Since the root node has no parent in T_H, we set it as *Null*. Additionally, we assume that the node-operator correctly sets the values for *Trace*, although in practice some additional bookkeeping has to be done to keep track of the jump timings.

Algorithm 2: Path analysis algorithm

```
Input  : An analysis path $p$.
Output: Reach($p$).
$R := \emptyset$;
$Q := \{\text{node}(\ell_0, \mathit{Init}, \mathit{Null})\}$;
while $Q \neq \emptyset$ do
    $N := \text{pop}(Q)$;                                  // Removes $N$ from $Q$
    $R := R \cup \left(\bigcup_{S \in \text{FP}(N)} S\right)$;
    if $\text{depth}(N) \leq |p|$ then
        for $(\ell', S') \in \text{JS}(N)$ do
            $N' := \text{node}(\ell', S', N)$;
            if $\mathit{isOnPath}(N', p)$ then
                $\text{push}(Q, N')$;                      // Adds $N'$ to $Q$
            end
        end
    end
end
return $R$;
```

Reachtrees and path analysis will be important tools for formalizing the CEGAR approach to handle urgency in Chapter 4. Before that we will extend hybrid automata with *urgent transitions* and describe how to apply the flowpipe-construction based analysis algorithm to this new class of automata in the next chapter.

Chapter 3

Urgent Hybrid Automata

After introducing hybrid automata in general and particularly the subclass of linear hybrid automata for which we described an analysis method we will next move on to the main focus of this thesis, which are *urgent hybrid automata*. The first step is to define urgent automata and their semantics in Section 3.1. We will also highlight their usefulness when modeling hybrid systems and reference related analysis approaches. Next, we will discuss the problem of computing the set difference of convex sets in Section 3.2, which is a key ingredient in the analysis method for urgent LHA we present in Section 3.3. Here we essentially extend the flowpipe-construction based algorithm from Section 2.3 to handle urgency. Finally we discuss a specialized analysis algorithm that can be used for automata with constant dynamics in Section 3.4.

3.1 Definition and Applications.

The general idea of urgency in hybrid systems is to enforce discrete change, which effectively restricts the set of states from which time is allowed to elapse. There are several approaches to formalize this idea for hybrid automata. Here we extend automata with a set of *urgent transitions* which impede time elapse as soon as they are enabled, as is done for example in [NÁW15]. Another common approach is to define *urgent locations* in which time is not allowed to elapse as soon as any discrete transition or some *urgency condition* is enabled [MF14].

Definition 3.1.1. An *urgent hybrid automaton* is a tuple

$$H = (\mathit{Loc}, \mathit{Var}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Urg}, \mathit{Init}),$$

such that $(\mathit{Loc}, \mathit{Var}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Init})$ is a hybrid automaton and $\mathit{Urg} \subseteq \mathit{Edge}$.

Subclasses such as urgent linear hybrid automata are defined analogously to subclasses of hybrid automata without urgency, in that they are extended by the set of urgent transitions Urg. The semantic difference between a non-urgent and an urgent transition is that if an urgent transition is enabled, time can no longer elapse. We therefore define the semantics for urgent hybrid automata as follows by the two rules:

$$\frac{e = (\ell, \mu, \ell') \in \mathit{Edge} \quad x, x' \in \mathbb{R}^d \quad (x, x') \in \mu \quad x' \in \mathit{Inv}(\ell')}{(\ell, x) \xrightarrow{e} (\ell', x')} \ \mathit{Rule}_{discrete}$$

$$\frac{\begin{array}{c} x, x' \in \mathbb{R}^d \quad \ell \in \mathit{Loc} \quad f \in \mathit{Flow}(\ell) \\ f(0) = x \quad f(t) = x' \quad \forall \varepsilon \in [0, t]. f(\varepsilon) \in \mathit{Inv}(\ell) \\ \forall \varepsilon \in [0, t).\ \forall e = (\ell, \mu, \ell') \in \mathit{Urg}.\ \left(\nexists x''.\ (\ell, f(\varepsilon)) \xrightarrow{e} (\ell', x'')\right) \end{array}}{(\ell, x) \xrightarrow{t} (\ell, x')} \ \mathit{Rule}_{time}$$

Note that $\mathit{Rule}_{discrete}$ is the same as for hybrid automata without urgency (see Definition 2.1.2) and that Rule_{time} is extended with the additional premise

$$\forall \varepsilon \in [0, t).\ \forall e = (\ell, \mu, \ell') \in \mathit{Urg}.\ \left(\nexists x''.\ (\ell, f(\varepsilon)) \xrightarrow{e} (\ell', x'')\right).$$

This premise says that as soon as any urgent transition is enabled, time can no longer elapse and thus a discrete transition must be taken. Note that (i) the time interval $[0, t)$ in which no urgent transition can be enabled is half open and does not include t, which means that an urgent transition can be enabled at exactly one point on a trajectory and (ii) multiple urgent transitions can be enabled at the same time, in which case any of them, or even some non-urgent transition can be taken.

Applications of urgency. Urgency can significantly help accurately modeling systems as hybrid automata and is useful whenever we want to enforce discrete change. As an intuitive example we consider again the vehicle modeled in Example 2.1.1. Here, we may want to add an urgent transition from the driving to braking state, so that the vehicle stops accelerating when going above a certain velocity threshold.

As another example consider a water tank in a factory that can only be drained as long as the filling level is high enough or other, more complex conditions are satisfied. Here, the draining process may also have to stop instantly, which can be enforced by adding urgency to the model. Such situations are common in practice when analyzing programmable logic controllers (PLCs), which are often used to control the behavior of plants, where the need for urgency can arise as explained in [NÁW15]. Here, PLCs are specified using sequential function charts (SFC) which can be defined using urgent semantics, i.e., steps in a SFC are only active as long as no transition is enabled [NÁ12]. These semantics are most accurately modeled as hybrid automaton by using urgent transitions.

Another source of urgency is the translation of deterministic simulation models used by modeling environments such as Simulink or Modelica to hybrid automata. This is an important application, because it is hard to cover all possible behaviors of a system using simulation. Verification of hybrid automata on the other hand allows to prove safety of a system, considering all possible input combinations. The derivation of hybrid automata from deterministic modeling languages is therefore a crucial step in verifying safety-critical systems. The translation of simulation models to hybrid automata has been explored in [SJ12] and [MF16a]. In [MF16b] the tool SL2SX is presented which automatically translates Simulink to SpaceEx models and makes use of urgent transitions in the resulting automaton.

Similarly, in [BMPW14, BMMW15], Bogomolov et al. apply hybrid model checking to planning problems by translating instances in the PDDL+ language to hybrid automata with urgency. PDDL+ [FL06] is an extension of the Planning Domain Definition Language (PDDL) which is widely used in the domain of artificial intelligence and has applications in e.g., robotics and embedded systems.

Related work. In the following sections we want to develop a reachability analysis algorithm for urgent automata, where we again focus on the subclass of urgent linear hybrid automata, which we denote by ULHA. We briefly highlight some related algorithms, some of which we will explore in more detail later as well.

One common approach to handling urgency in hybrid systems is explored in [SJ12] and [BMMW15]. The goal here is to construct an equivalent automaton without urgency, for which one of the classical reachability algorithms can be used. The idea to do this is to replace each location with an outgoing urgent transition by multiple new locations whose invariants are extended by the inverted halfspaces of the guard. This ensures that time can only elapse as long as the interior of the guard is not entered, however some over-approximation is introduced by trajectories that touch the boundary of the guard. In [BMMW15] a way to minimize this over-approximation is presented. Here we discuss this issue in more detail in Section 3.3.

An algorithm for handling urgency in the class of *rectangular automata* is given in [MF14]. Here, urgency is realized by an urgency condition, which when satisfied impedes time elapse. To perform reachability analysis, the complement of the urgency condition is added as invariant, which reduces the problem to automata with non-convex invariants. The algorithm has been implemented in the tool PHAVer and later been extended to affine hybrid automata [MF16a]. We discuss the latter algorithm in more detail in Section 3.3 and present an analysis algorithm for LHA I in Section 3.4 which is closely related to the approach in [MF14].

3.2 Set Difference Computation

A central part of the reachability algorithm for linear hybrid automata will be the computation of the *set difference* operator, which is defined as

$$P \setminus Q \coloneqq \{x \in P \mid x \notin Q\}, \tag{3.1}$$

for sets $P, Q \subseteq \mathbb{R}^d$. In this section we outline an algorithm for computing the set difference between polytopes, however the implementation and optimization of the operation is beyond the scope of this thesis. More details and in particular the implementation of the set difference operation used in later parts of this thesis are developed in [Amf21].

We now consider the case where P and Q are convex polyhedra and want to obtain $P \setminus Q$. The main difficulty is that the result of Equation (3.1) is not necessarily a convex set again, which can be seen at the example of intervals where $[0,3] \setminus [1,2] = [0,1) \cup (2,3]$ is not convex.

To circumvent this problem, the simplest idea is to use the convex hull of the set difference, which can in some cases be sufficient, but may not be precise enough in general. In particular, we may end up with the original set, as in the example above $\mathrm{chull}([0,1) \cup (2,3]) = [0,3]$.

A more precise result can be obtained by computing the set difference as a union of a finite number of polyhedra. Let $P, Q \subseteq \mathbb{R}^d$ be closed polyhedra, where Q is defined by the inequalities $\bigwedge_{i=1}^{n} c_i^T x \leq d_i$. We follow the approach presented in [BMDP02] and define the sets

$$\begin{aligned} R_1 &:= \{x \in P \mid c_1^T x > d_1\} \\ R_i &:= \{x \in P \mid c_i^T x > d_i \wedge \forall j \in \{1, \dots, i-1\}.\ c_j^T x \leq d_j\}, \end{aligned} \tag{3.2}$$

for $i = 2, 3, \dots, n$. Then $P \setminus Q = \cup_{i=1}^{n} R_i$. We briefly illustrate an example of this algorithm in Figure 3.1.

In [RKML06] this algorithm is extended to set differences of unions of polyhedra $\mathcal{P}$ and $\mathcal{Q}$. We write $\mathcal{P} = \cup_{i=1}^{m_\mathcal{P}} P_i$ and $\mathcal{Q} = \cup_{j=1}^{m_\mathcal{Q}} Q_j$ where the P_i and Q_j are the convex components of $\mathcal{P}$ and $\mathcal{Q}$ respectively. To compute $\mathcal{P} \setminus \mathcal{Q}$ we can then use the identities

$$\left(\bigcup_{i=1}^{m_\mathcal{P}} P_i\right) \setminus \mathcal{Q} = \bigcup_{i=1}^{m_\mathcal{P}} (P_i \setminus \mathcal{Q})$$

$$P_i \setminus \left(\bigcup_{j=1}^{m_\mathcal{Q}} Q_j\right) = (((P_i \setminus Q_1) \setminus Q_2) \cdots) \setminus Q_{m_\mathcal{Q}},$$

to reduce the problem back to computing set difference for polyhedra.

In [Bao05, Bao09] this approach is improved by giving a *branch-and-bound* algorithm for computing the set difference for unions of polyhedra directly. The idea here is to more efficiently find the feasible constraint combinations in Equation (3.2), i.e., find the non-empty polyhedra faster.

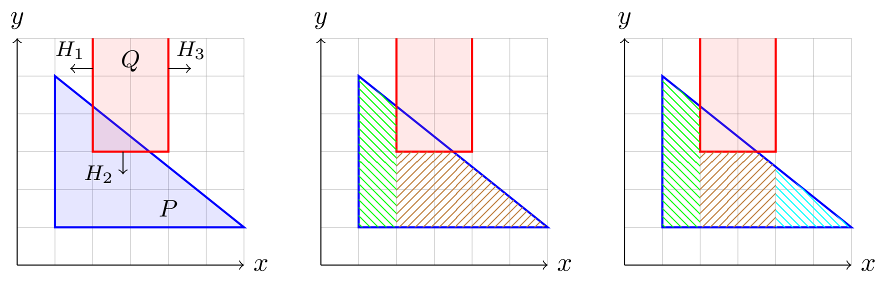
Number of components. When using the set difference operation in reachability analysis (see Section 3.3) and obtaining a covering $\cup_{i=1}^{n} R_i$ of $P \setminus Q$ we will continue the computation with each component R_i individually. This can cause a large blowup in complexity, especially when set difference has to be computed with multiple components, so we are interested in an upper bound on the number of non-empty components R_i. As a general setting we consider a polyhedron P and a set $\mathcal{Q}$, which is the union of polyhedra $Q_1, \dots, Q_{m_\mathcal{Q}}$ and we use $|Q_i|$ to denote the number of inequalities defining Q_i.

If we repeatedly apply Equation (3.2) to compute the set difference $P \setminus \mathcal{Q}$, we get $|Q_1|$ components in the first iteration. For each of those we compute the set difference with Q_2 which gives $|Q_2|$ new components each. Continuing this we end up with

$$\prod_{i=1}^{m_\mathcal{Q}} |Q_i|$$

components that represent the set difference $P \setminus \mathcal{Q}$. However, as noted in [Bao05, Bao09] some of these components are bound to be empty. An upper bound on the number of components that can be non-empty is related to a hyperplane arrangement problem which asks how many regions $\mathbb{R}^n$ can be divided into by a set of hyperplanes. This question is studied in [Buc43] and is used in [Bao09] to bound the number of non-empty regions R_i by

$$\sum_{j=1}^{d} \binom{M}{j} = \mathcal{O}\left(\frac{M^d}{d!}\right),$$



(a) Polytopes P and Q with defining halfspaces.
(b) Order: H_1, H_2, H_3
(c) Order: H_1, H_3, H_2

Figure 3.1: Components when using different orderings of the halfspaces for computing the set difference $P \setminus Q$ shown in (a). The halfspaces H_1, H_2, H_3 define Q. When the order in (b) is used, the result has two components while in (c) it has three.

where $M = \sum_{i=1}^{m_Q} |Q_i|$ is the total number of constraints. This bound is especially useful in lower dimension when $\mathcal{Q}$ consists of multiple components with many constraints.

In practice when computing the set difference of two polytopes $P \setminus Q$ it can be very relevant in which order the halfspaces defining Q can be considered. An example for this can be seen in Figure 3.1, where a bad ordering of the halfspaces results in more components than necessary.

Over-approximation. A practical issue for computing the set difference is that we have used strict inequalities in Equation (3.2) but in practice often work with representations that can only represent closed sets (such as for example $\mathcal{V}$-polytopes). To over-approximate the result we may therefore replace the strict inequalities in Equation (3.2) with weak ones. Note that this is indeed an over-approximation, because if $c^T x < d$ then also $c^T x \leq d$, i.e., we get a superset of the set difference.

An advantage is that in our reachability algorithm (see Section 3.3) we usually want to compute the closure $\mathrm{cl}(P \setminus Q)$ of the set difference which is the same as the proposed over-approximation except for empty components. For empty components we may indeed over-approximate the closure if we replace strict inequalities with weak ones, which can be seen on the example $R = \{x \in \mathbb{R} \mid x < 0 \wedge x \geq 0\}$. Obviously, $R = \emptyset$, but if we replace the strict inequality $x < 0$ in R, we get $\{x \in \mathbb{R} \mid x \leq 0 \wedge x \geq 0\} = \{0\} \neq \mathrm{cl}(R)$. It may therefore be useful to first check emptiness of each component (with strict inequalities) and only if it is non-empty compute the closure by replacing the strict inequalities.

Boxes. Lastly we want to mention a specialized algorithm which can be used when using boxes (Cartesian products of intervals) as state set representation. In particular, [JCKK18] present an algorithm for computing the set difference of two boxes $A, B \subseteq \mathbb{R}^d$. The algorithm produces $2d$ boxes (some of which may be empty) whose union represent the set difference $A \setminus B$. While this is the same number of components created by applying the set difference algorithm with polytopes (B has $2d$ defining inequalities as a polytope), the boxes are constructed directly without redundant inequalities which may make the algorithm easier to implement and optimize in practice.

A technical difficulty when using boxes as a state set representation for reachability analysis arises from the conversion of sets that are not box shaped: To illustrate, assume we want to compute the set difference of two sets $P \setminus Q$. The straightforward approach is to obtain box representations P_B, Q_B for P and Q and apply the set difference algorithm to P_B and Q_B. The issue is that the box representation is typically over-approximative, i.e., $P_B \supseteq P$ and $Q_B \supseteq Q$, as explained in Section 2.2. Using an over-approximative conversion for Q is problematic, because the result $P_B \setminus Q_B$ is in general not over-approximative for $P \setminus Q$. One approach to circumvent this is to underapproximate Q by one or multiple boxes. Some ideas for such underapproximations can be found in [BFT04]. Here, we instead restrict application of the set difference algorithm for boxes to cases where Q is already box shaped in which case the representation Q_B is exact. In all other cases we compute the set difference using the polytope based algorithm and convert the resulting polytopes back to boxes.

3.3 Utilizing Set Difference

Throughout this section we fix an urgent linear hybrid automaton (ULHA) H and the goal is to compute the set of reachable states of H. We mainly focus on how the time successors of an initial state set up to a time bound T can be computed by adapting the flowpipe-construction detailed in Section 2.3. Note that the discrete semantics for urgent hybrid automata are the same as for hybrid automata without urgency. It is therefore sufficient to compute the set of reachable states by letting time elapse and compute the discrete successors analogously to regular LHA as described in Section 2.3.

For each urgent transition $e = (\ell, g, r, \ell')$ we define the *jump enabling set*

$$J_e \coloneqq g \cap r^{-1}(\mathit{Inv}(\ell')), \tag{3.3}$$

where $r^{-1}(X)$ denotes the *pre-image* of a set $X \subseteq \mathbb{R}^d$, defined as $r^{-1}(X) \coloneqq \{x \in \mathbb{R}^d \mid r(x) \in X\}$. Note that if $x \in J_e$ then $x \xrightarrow{e} r(x)$, which follows directly from the definition of the discrete semantics. We further define

$$J_\ell \coloneqq \bigcup_{(\ell,g,r,\ell') \in \mathit{Urg}} J_e,$$

as the union of the jump enabling sets for $\ell \in \mathit{Loc}$.

Assume now that we are given an initial state set (ℓ, X_0), a time bound T and a step size $\delta = \frac{T}{n}$ for some $n \in \mathbb{N}$. We assume that all initial states satisfy the invariant, i.e., $X_0 \subseteq \mathit{Inv}(\ell)$. Since H is a LHA, the set of flow functions in ℓ can be given as the solution set of an ODE

$$\dot{x} = Ax. \tag{3.4}$$

Recall from Section 2.3 that solutions to eq. (3.4) are of the form $x(t) = e^{\delta A}x(0)$. We construct the first segment Ω_0 analogously to regular flowpipe-construction by applying the affine transformation $e^{\delta A}$ to X_0, bloating the result and taking the convex hull with X_0.

$$\Omega_0 = \mathrm{chull}(X_0 \cup (e^{\delta A} X_0 \oplus \mathcal{B})).$$

To account for the invariant and the urgent transitions we define

$$\Omega_0' = \mathrm{cl}((\Omega_0 \cap \mathit{Inv}(\ell)) \setminus J_\ell),$$

The following segments Ω'_i for $i = 1, \ldots, n-1$ are defined by the recurrence relation

$$\begin{aligned} \Omega_i &= e^{\delta A}\Omega'_{i-1}, \\ \Omega'_i &= \mathrm{cl}((\Omega_i \cap \mathit{Inv}(\ell)) \setminus J_\ell). \end{aligned} \tag{3.5}$$

The proposed algorithm for computing time successors consists of successively computing the constrained segments Ω'_i. We show correctness of the algorithm by proving the following lemma:

Lemma 3.3.1. *Let* $R_{[0,T]} = \{x \in \mathbb{R}^d \mid \exists t \in [0,T].\ \exists x_0 \in X_0.\ (\ell, x_0) \xrightarrow{t} (\ell, x)\}$ *be the set of reachable states in the time interval* $[0,T]$ *and let* Ω'_i *be defined as above. Then*

$$R_{[0,T]} \subseteq X_0 \cup \bigcup_{i=0}^{n-1} \Omega'_i.$$

Proof. We first show that $R_{[0,\delta]} \subseteq X_0 \cup \Omega'_0$. By construction of Ω_0 it holds that $R_{[0,\delta]} \subseteq \Omega_0$. Since states that do not satisfy the invariant are by definition not reachable we have that $R_{[0,\delta]} \subseteq \Omega_0 \cap \mathit{Inv}(\ell)$.

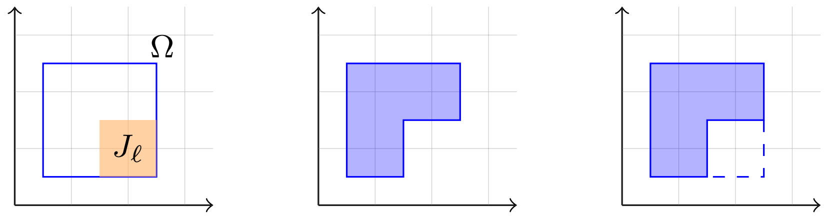
Now let $x \in R_{[0,\delta]}$. Then for some $f \in \mathit{Flow}(\ell)$ and a time point $t \in [0,\delta]$ the identity $f(t) = x$ holds. If $t = 0$ then $x \in X_0$ and we are done, so we assume in the following that $t > 0$. We want to show that $x \in \Omega'_0$ which is equivalent to showing that for all $\varepsilon > 0$,

$$\mathcal{B}_x(\varepsilon) \cap ((\Omega_0 \cap \mathit{Inv}(\ell)) \setminus J_\ell) \neq \emptyset. \tag{3.6}$$

Assume to the contrary that for some $\varepsilon > 0$ Equation (3.6) does not hold, i.e., the left hand side is empty. By continuity of f there exists some $t' \in [0,t)$ such that the distance between $x = f(t)$ and $f(t')$ is smaller than ε, i.e., $f(t') \in \mathcal{B}_x(\varepsilon)$. Since Equation (3.6) is assumed to not hold and because $f(t') \in R_{[0,T]} \subseteq \Omega_0 \cap \mathit{Inv}(\ell)$ it follows that $f(t') \in J_\ell$, which means that at time t' some urgent jump is enabled. By definition of the urgent semantics it follows that from $f(t')$ time cannot elapse, and so x is in fact not reachable, in contradiction to $x \in R_{[0,\delta]}$. Therefore, we get by proof of contradiction that $x \in \Omega'_0$. Since this argument holds for all $x \in R_{[0,\delta]}$, it follows that $R_{[0,\delta]} \subseteq X_0 \cup \Omega'_0$.

Note that we only needed X_0 to contain valuations at time 0 and we showed that Ω'_0 contains all valuations in the time interval $(0,\delta]$. Thus it follows by induction for $i > 0$ that $R_{(i\cdot\delta,(i+1)\cdot\delta]} \subseteq \Omega'_i$. In conclusion, by taking X_0 and all Ω'_i, we get a covering of the whole time interval $[0,T]$ which proves the lemma. □

A similar approach to compute time successors with urgent semantics is described in [MF16a]. Here, an *urgency condition* $U(\ell)$ is given as a (possibly non-convex) subset of $\mathbb{R}^d$ for every location ℓ. Semantically, when a state satisfies the urgency condition, time can no longer elapse so in our setting $U(\ell) = J_\ell$. In [MF16a], the closure of the complement $\mathrm{cl}(\overline{U(\ell)})$ is added to the invariant of ℓ and a way to compute time successors with non-convex invariants is described. While this is equivalent to our approach in most cases, there are some cases where intersecting with the closure of the complement of J_ℓ would give a slightly larger result (see for example Figure 3.2).



(a) Segment Ω and jump enabling set J_ℓ (b) $\mathrm{cl}(\Omega \setminus J_\ell)$ (c) $\Omega \cap \mathrm{cl}(\overline{J_\ell})$

Figure 3.2: Set difference with J_ℓ in (b) and intersection with the complement of the closure of J_ℓ in (c), which additionally contains the dashed edges.

Computing the segments. Now that we have described an approach to construct time successors for urgent LHA, we now want to focus on how they can be computed in practice. First, we need to obtain the set J_ℓ of jump enabling sets, which comes down to computing the sets J_e for outgoing urgent transitions of ℓ (see Equation (3.3)). The most interesting part of eq. (3.3) is the set $r^{-1}(Inv(\ell'))$ which is pre-image of the invariant in the target location ℓ' under the reset function r. Recall that we are working with LHA, where the invariant in ℓ' is a convex closed polytope $Ix \leq c$, with $I \in \mathbb{R}^{n\times d}$ and $c \in \mathbb{R}^n$, given as the matrix-vector pair (I, c). The reset function on the other hand is given as an affine transformation $x' = Ax + b$ with $A \in \mathbb{R}^{d\times d}$ and $b \in \mathbb{R}^d$.

We compute the set $r^{-1}(Inv(\ell'))$ as follows:

$$\begin{aligned} r^{-1}(Inv(\ell')) &= \{x \in \mathbb{R}^d \mid r(x) \in Inv(\ell')\} \\ &= \{x \in \mathbb{R}^d \mid Ir(x) \leq c\} \\ &= \left\{x \in \mathbb{R}^d \mid I\,(Ax + b) \leq c\right\} \\ &= \left\{x \in \mathbb{R}^d \mid IAx \leq c - Ib\right\}. \end{aligned}$$

This is again a convex closed polytope defined by the matrix-vector pair $(IA, c - Ib)$. To obtain J_e we only need to compute the intersection with the guard g, which is easily done because g is also given as a closed convex polytope.

Next we use the set difference operation in eq. (3.5) to obtain the sets Ω_i'. Both input sets Ω_i as well as J_ℓ are unions of polyhedra, for which we have described a way to compute the set difference in Section 3.2. Our approach to computing the flowpipe with urgent semantics is summarized in Algorithm 3. Here, we use the term *fragment* to describe the multiple components of a segment that can be created by the set difference operator. In particular, the fragments of a segment are used to cover the reachable states in the same time interval. In Algorithm 3 we use the function "setdifference" which returns a set of fragments representing the closure of the set difference.

Algorithm 3: Flowpipe time successor computation for urgent LHA

Input : A location ℓ with flow $\dot{x} = Ax$, an initial set $X_0 \subseteq Inv(\ell)$, a time bound T and a time step $\delta = \frac{T}{n}$.

Output: Flowpipe of segments whose union contains the set of reachable states from X_0 within the time bound T.

```
$FP := \{X_0\}$;
$\Omega_0 :=$ firstSegment$(X_0, \delta) \cap Inv(\ell)$;
/* setdifference creates multiple (closed) fragments */
$previousFragments =$ setdifference$(\Omega_0, J_\ell)$;
$previousFragments =$ removeEmptyFragments$(previousFragments)$;
$FP := FP \cup previousFragments$;
for $i \leftarrow 1$ to $n$ do
    if $previousFragments = \emptyset$ then
        return $FP$;
    end
    $nextFragments := \emptyset$;
    for $\Omega^j_{i-1} \in previousFragments$ do
        $nextFragments := nextFragments \cup \mathrm{cl}\left(\left(e^{\delta A}\Omega^j_{i-1} \cap Inv(\ell)\right) \setminus J_\ell\right)$;
    end
    $nextFragments :=$ removeEmptyFragments$(nextFragments)$;
    $FP := FP \cup nextFragments$;
    $previousFragments := nextFragments$;
end
return $FP$;
```

Shadow of J_ℓ. While the descibed method to compute time successors for ULHA is correct, it can introduce additional over-approximation. The first source of over-approximation can be described as the “shadow” of the jump enabling set J_ℓ that may not be correctly excluded from the computed set of reachable states. The shadow of J_ℓ is the set of states that is only reachable from the initial state set by trajectories that pass through J_ℓ. In the computation of a single segment however, only the interior of J_ℓ is excluded by computing the set difference and states that lie in the shadow of J_ℓ may not be detected. Figure 3.3 illustrates this complication on two examples. Here, the first segment Ω_0 is constructed from an initial set X_0 and then the set $\mathrm{cl}(\Omega_0 \setminus J_\ell)$ is computed. In (a) it is clear that the shaded states are not reachable from X_0 because any trajectory from X_0 must pass through J_ℓ. In (b) the situation is somewhat less clear, because in principal some non-linear trajectory could move around J_ℓ to reach a state in the shaded area. If we for simplicity assume a linear flow as indicated by the arrow, then the shaded area is again not reachable.

Detecting states that lie in the shadow of J_ℓ and are therefore unreachable is hard in a general setting. In Section 3.4 we describe methods that work for urgent LHA I, i.e., for automata where the dynamics are constant. Here we briefly sketch an approach that can in some cases be used to check whether a fragment computed by the set difference operation is entirely unreachable, which can in particular detect situations such as Figure 3.3a.

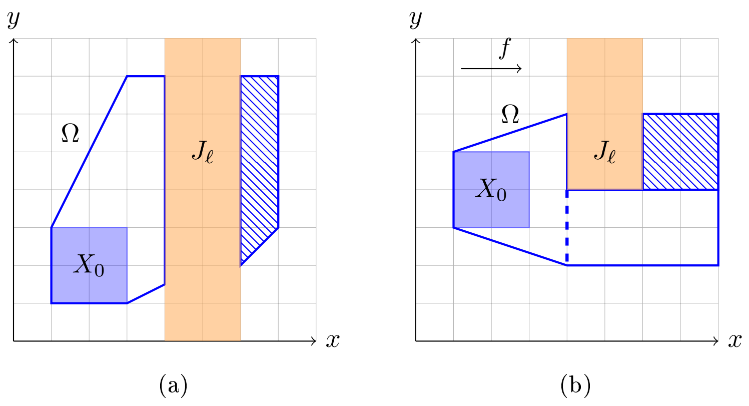


Figure 3.3: The shadow of J_ℓ can introduce additional over-approximation. In (b) the direction of the linear flow is indicated by the arrow. The shaded fragments are contained in the computed set $\mathrm{cl}(\Omega \setminus \overline{J_\ell})$ but are not reachable.

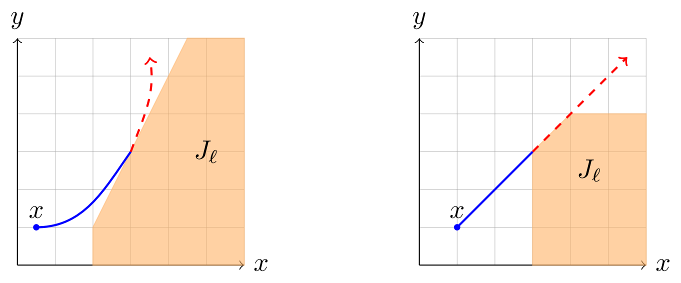
For $i \geq 0$ let $\Omega_i^1, \ldots, \Omega_i^{n_i}$ be a family of convex closed sets that covers $R_{(i\cdot\delta,(i+1)\cdot\delta]}$ as computed by Algorithm 3. For ease of notation we here write $\Omega_{-1}^1 = X_0$ for the single fragment that covers the initial states. If a fragment Ω_i^k is reachable then by continuity of the flow functions there must be a trajectory from some previous fragment Ω_{i-1}^l to Ω_i^k that only moves through fragments

$$\Omega^1, \Omega^2, \Omega^3, \ldots, \Omega^m, \tag{3.7}$$

where $\Omega^1 = \Omega_{i-1}^l$ and $\Omega^m = \Omega_i^k$ and the fragments Ω^j belong to the time interval $(i\cdot\delta,(i+1)\cdot\delta]$, i.e., $\Omega^j = \Omega_i^{p_j}$ for some indices $p_j \in \{1,\ldots,n_i\}$ and $2 \leq j \leq m$. This means that the intersection $\Omega^j \cap \Omega^{j+1}$ must be non-empty for all $j \geq 1$, so that a continuous trajectory can move through them. To check for which fragments Ω_i^k such a trajectory exists, which is necessary for them to be reachable, we can build a graph where the fragments Ω_i^k are the nodes with an edge between them if the intersection of the corresponding fragments is non-empty. The nodes whose fragments have a non-empty intersection with some previous fragment Ω_{i-1}^l are designated as starting nodes. If we look at the connected components of the constructed graph, then only the fragments whose node belongs to a component with a starting node are potentially reachable. For the other fragments on the other hand no sequence such as described in Equation (3.7) exists and they are certainly not reachable.

In Figure 3.3a the tree for $i = 0$ has two nodes corresponding to the fragments left and right of J_ℓ respectively. The nodes are not connected, because the fragments have empty intersections and the node corresponding to the left fragment is a starting node, as it has non-empty intersection with X_0. Since there is no path from a starting node to the node corresponding to the right fragment, it can be discarded as it is unreachable. Note that the existence of such a path is only necessary and not sufficient. As an example consider the splitting indicated by the dashed lines in Figure 3.3b. Then the graph for $i = 0$ has three nodes in a single connected component i.e., no fragments can be discarded although the shaded fragment is in fact not reachable.

Note that the over-approximation caused by the shadow of J_ℓ can in a lot of cases be mitigated by choosing a sufficiently small time step. In Figure 3.3a for example, if the time step is chosen sufficiently small no computed segment will include points



(a) Non-linear trajectory (b) Linear trajectory

Figure 3.4: Trajectories that touch the boundary of J_ℓ but do not enter the interior. In both examples the dashed (red) part of the trajectory is not reachable.

from both left and right of J_ℓ. Instead, some segment may be contained completely in J_ℓ, at which point the computation stops because the computed set difference is empty.

In particular, the shadow of J_ℓ becomes a major problem when the time successor computation is done in a single iteration, which is possible for automata with constant dynamics. We discuss this case in more detail in Section 3.4.

Touching the boundary of J_ℓ. The second source of over-approximation we want to bring up is caused by trajectories that touch the boundary of J_ℓ but don't move on to its interior. An example for this behavior is shown in Figure 3.4a. From an initial point x, the trajectory hits the boundary of J_ℓ at which point the semantics forbid time to elapse further. The points that lie on the dashed part of the trajectory are therefore not reachable from x, which is undetected by our flowpipe-construction algorithm. Note that this problem is not limited to non-linear dynamics, as can be seen in Figure 3.4b where again time can not longer elapse after the boundary J_ℓ has been touched and the points on the dashed line are not reachable.

Detecting this behavior for non-linear dynamics seems even harder than identifying points in the shadow of a guard since we have no way to notice at which point J_ℓ was first reached when using a geometric approach. In [BMMW15], where urgency is handled by constructing auxiliary locations whose invariants are extended by the inverted halfspaces of the guard, this problem is mitigated by introducing an additional clock t. The variable t ensures that after the guard has been satisfied once, less than $\varepsilon > 0$ more time units can be spent in ℓ, which means that with a sufficiently small ε the problem of touching the guard and moving out of it again is almost eliminated.

3.4 Urgent LHA I

We now consider a special case of reachability analysis with urgent transitions, which is the subclass of *urgent LHA I*. Recall that LHA I are automata with constant dynamics, i.e., in every location ℓ, the dynamics are given by an ODE of the form

$$\dot{x} = f,$$

where $f \in \mathbb{R}^d$.

For LHA I the time successors of an initial state set (ℓ, X_0) can be computed exactly in one step [ACH+95] as

$$T^+ = (\ell, X_0 \oplus F \cap \mathit{Inv}(\ell)),$$

where $F := \{tf \mid t \in [0,T]\}$ is the cone generated by the flow vector f, and $\oplus$ is the Minkowski-sum operator, defined in Section 2.2. The jump successors J^+ of the computed state set T^+ can be computed [ACH+95] with the formula

$$J^+ = \bigcup_{(\ell,g,r,\ell')\in \mathit{Edge}} (\ell', r(T^+ \cap g) \cap \mathit{Inv}(\ell')),$$

which implies that the bounded reachability problem is decidable for LHA I.

In this section we want to outline an approach how to compute the reachable states for LHA I with urgent transitions, i.e., without over-approximation. This is particularly important, because if we were to apply the same approach as described in Section 3.3 and simply take $\mathrm{cl}(T^+ \setminus J_\ell)$ as the set of constrained time successors, we can potentially get a large over-approximation. This is caused by the shadow of J_ℓ, which as already mentioned becomes more problematic the larger the selected time step size is and in the case of LHA I the time step is essentially equal to T.

A closely related method to the one we presented here is described in [MF14]. It works even for *rectangular automata*, in which the dynamics may be given as an interval of possible derivatives for each variable. We will first describe our approach and then compare it with the algorithm from [MF14].

We start with an initial state set (ℓ, X_0) and our goal is to exactly compute the set of reachable states by letting time elapse and taking a single discrete transition. By applying this method iteratively, the reachable states can be computed up to a given jump depth. To simplify the problem we first assume that the initial location ℓ has exactly one outgoing urgent transition e and will later discuss how the method can be extended to multiple urgent transitions. For ease of notation we also assume that $X_0 \subseteq \mathit{Inv}(\ell)$, and otherwise consider $X_0 \cap \mathit{Inv}(\ell)$ as initial set.

Jump predecessors. The first step of our approach is to compute the jump predecessors J_e^-, i.e., the set of states from which the transition e can be taken. Since e is urgent, these states are the time successors of X_0 where J_e is first entered:

$$J_e^- = \{x(t) \mid x \in X_0 \wedge x(t) \in \mathit{Inv}(\ell) \wedge x(t) \in J_e \wedge \forall t' < t.\ x(t') \notin J_e\},$$

where we use the notation $x(t) = x + ft$ for $t \geq 0$ to describe the time successor of some $x \in \mathbb{R}^d$ after letting t time units elapse. To construct J_e^- we consider each halfspace of J_e independently and take for each halfspace H the states where H is first satisfied. Formally, let H be a halfspace defined by the inequality $a^T x \leq b$ for $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$. We define the sets $H^\sim = \{x \in \mathbb{R}^d \mid a^T x \sim b\}$ for $\sim \in \{<, \leq, =, \geq, >\}$, and in particular $H = H^\leq$. The set where H is first entered from X_0 can be described by the equation

$$\mathit{entry}(H) = \big((X_0 \cap H^>) \oplus F\big) \cap H^=.$$

Intuitively, we want H to not be satisfied initially and capture the point in time where it is first satisfied. By linearity of the flow and convexity of H, the entry point must be on the border $H^=$. Note that $\mathit{entry}(H)$ does not include states that are initially in J_e, so we will have to handle $X_0 \cap J_e$ separately. We summarize our results in the following lemma.

Lemma 3.4.1. *Let J_e be defined as the intersection of halfspaces $J_e = H_1 \cap \cdots \cap H_n$. We define the set*

$$entry(J_e) = \bigcup_{i=1}^{n} \left(entry(H_i) \cap Inv(\ell) \cap J_e \right).$$

Then $J_e^- = entry(J_e) \cup (X_0 \cap J)$.

Proof. We first prove the inclusion $J_e^- \subseteq entry(J_e) \cup (X_0 \cap J_e)$. For that let $x' \in J_e^-$. Since $J_e^- \subseteq J_e$, it follows that if $x' \in X_0$ then $x' \in X_0 \cap J_e$ and we are done. We therefore assume without loss of generality that $x' \notin X_0$. Then there exists some initial state $x \in X_0$ such that $x' = x(t) = x + tf$ for some $t > 0$. We make the following observation:

- The premise $x(t) \in J_e$ is equivalent to $x(t) \in H_i = H_i^{\leq}$ for all i. If it were true that $x(t) \notin H_i^{=}$ for all i, then $x(t) \in H_i^{<}$ for all i, which together with the fact that the sets $H_i^{<}$ are open means that there is some $t' < t$ such that $x(t') \in J_e$. But then $x(t)$ is not reachable from x at all since the urgent jump was enabled at an earlier time t'. Thus there exists some i such that $x(t) \in H_i^{=}$. In the following we denote the indices for which this holds by $i_1, \ldots, i_k$, i.e., $x(t) \in H_{i_j}^{=}$ for $j = 1, \ldots, k$.

We want to show that $x(t) \in entry(H_{i_j}) \cap J_e$ for some $j \in \{1, \ldots, k\}$. Since $J_e^- \subseteq J_e$, we have $x(t) \in J_e$. By selection of the halfspaces H_{i_j} it also holds that $x(t) \in H_{i_j}^{=}$ for all j. It therefore remains to show that $x(t) \in \left(X_0 \cap H_{i_j}^{>} \right) \oplus F$ for some j, for which it is sufficient to prove that the initial value $x \in H_{i_j}^{>}$ for some j.

Assume to the contrary that $x \in H_{i_j}^{\leq}$ for all j. Since also $x(t) \in H_{i_j}^{=} \subset H_{i_j}^{\leq}$ it follows by convexity that $x(t') \in H_{i_j}^{\leq}$ for all $t' < t$. For all other halfspaces, for which $x(t) \in H_i^{<}$ must hold, we can find some smaller value t' such that $x(t') \in H_i^{<}$ for all i. This is because the sets $H_i^{<}$ are open and the flow is continuous. It follows that for some $t' < t$ we have that $x(t') \in J_e$, which means that again $x(t)$ is not reachable from x.

We have derived a contradiction from the assumption that $x \in H_{i_j}^{\leq}$ for all $j \in \{1, \ldots, k\}$. Thus $x \in H_{i_j}^{>}$ for some j which as stated above proves that $x(t) \in entry(J_e)$. Since $x' = x(t)$ was arbitrarily chosen from J_e^- the inclusion follows.

For the other inclusion we only need to show that

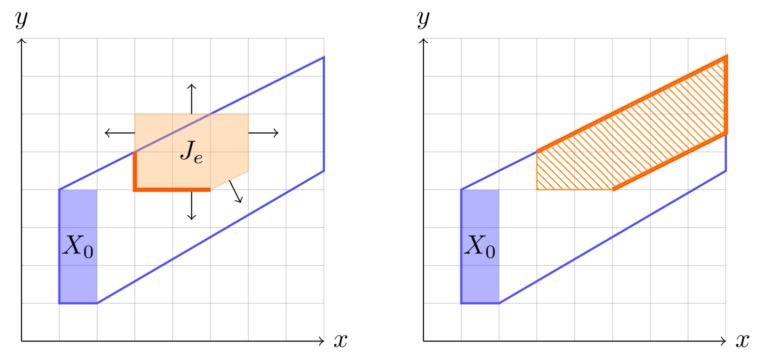
$$entry(H_i) \cap Inv(\ell) \cap J_e \subseteq J_e^-, \tag{3.8}$$

for all halfspaces H_i. The fact that $X_0 \cap J_e \subseteq J_e^-$ follows directly from the semantics because it is allowed to jump immediately after entering a location.

Let $H \in \{H_1, \ldots, H_n\}$ be one of the halfspaces defining J_e and

$$x' \in entry(H) \cap Inv(\ell) \cap J_e.$$

Then by definition of $entry(H)$ there exists an $x \in X_0 \cap H^{>}$ and a $t > 0$ such that $x' = x(t)$. By linearity of the flow it holds that $x(t') \in H^{>}$ for all $t' < t$. In particular, this means that $x(t') \notin J_e$, or in other words, $x(t) \in J_e$ is the first time successor of x at which e is enabled. Thus, $x(t) \in J_e^-$ and consequently eq. (3.8) holds for H. Since H was an arbitrary halfspace we get that $entry(J_e) \subseteq J_e^-$, which proves the lemma. □



(a) Jump predecessors $\mathit{entry}(J_e)$ (b) The shadow $\mathit{shadow}(\mathit{entry}(J_e))$

Figure 3.5: Unconstrained time successors of an initial state set X_0 and a jump enabling set J_e with its defining halfspaces in (a). The set of entry points $\mathit{entry}(J_e)$ consists of the bold orange lines. In (b) the corresponding shadow $\mathit{shadow}(\mathit{entry}(J_e))$ is indicated by the orange hatched area. Note that $\mathit{entry}(J_e)$ is not included in $\mathit{shadow}(\mathit{entry}(J_e))$.

Excluding the shadow. Now that we have obtained the states from which e can be taken, we need to make sure that time cannot elapse after that. In other words, we want to exclude the shadow of J_e^- which is the set

$$\mathit{shadow}(J_e^-) \coloneqq J_e^- \oplus \{ft \mid t > 0\}.$$

Note that we only allow values $t > 0$, which means that J_e^- itself will not be excluded. The set of time successors is then given by

$$T^+ = ((X_0 \oplus F) \cap \mathit{Inv}(\ell)) \setminus \mathit{shadow}(J_e^-).$$

To see that $R_{[0,T]} = T^+$, note first that $R_{[0,T]} \subseteq (X_0 \oplus F) \cap \mathit{Inv}(\ell)$ and that such a state $x(t)$ is reachable if and only if $x(t') \notin J_e^-$ for all $t' < t$, or equivalently, $x(t) \notin \mathit{shadow}(J_e^-)$. An example for the computation of the jump predecessors and the corresponding shadow is depicted in Figure 3.5.

Finally, the jump successors of the urgent transition e can be obtained by applying the reset to J_e^-. For all other jumps, we can intersect T^+ with the guard and apply the reset analogously to the case without urgency to obtain J^+. For the case that there is one urgent transition we have therefore described a way to compute the reachable states without over-approximation. The main problem when applying this method to multiple urgent transitions is that the jump enabling sets may lie in the shadows of each other. One idea to solve this is to compute the sets $\mathit{entry}(J_{e_i})$ for all urgent transitions independently and exclude the shadows of all other urgent transitions, i.e., take the set

$$\mathit{entry}(J_{e_i}) \setminus \left(\bigcup_{j \neq i} \mathit{shadow}\left(\mathit{entry}(J_{e_j}) \cup \left(J_{e_j} \cap X_0\right)\right) \right).$$

Note that we take the set difference with the shadows of the unconstrained sets $\mathit{entry}(J_{e_j})$, which may not all be jump enabling states, because they may lie in the shadow of another urgent jump. To see that this is correct assume that we have

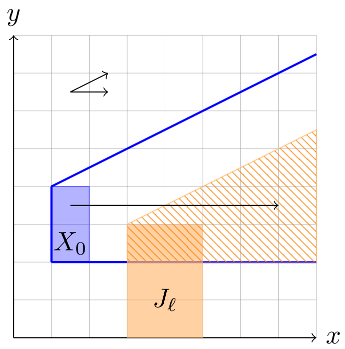


Figure 3.6: An example where states in the shadow of $\mathit{entry}(J_\ell)$ are reachable when rectangular dynamics are present. The flow is indicated by the arrows in the top left corner and a trajectory from the initial set X_0 to the shadow of $\mathit{entry}(J_\ell)$ (orange hatched area) is given by the large arrow.

three urgent jumps e_1, e_2 and e_3. If some subset $P \subseteq \mathit{entry}(J_{e_1})$ lies in the shadow of $\mathit{entry}(J_{e_2})$, then even if $\mathit{entry}(J_{e_2})$ lies completely in the shadow of $\mathit{entry}(J_{e_3})$, which means that it is not a jump enabling set at all, then also $P \subseteq \mathit{shadow}(\mathit{entry}(J_{e_3}))$. This means that taking the set difference with $\mathit{shadow}(\mathit{entry}(J_{e_2}))$ is still correct.

As a subject of future work it could be interesting to explore whether the sets of jump predecessors can be computed directly without taking the pairwise set differences.

A related approach. In [MF14] a similar approach to exactly compute reachable states for LHA I with urgency is presented. Here, an urgency condition U is assumed, which when satisfied forbids further time to elapse. For the sake of time successor computation, U plays the same role as the set of jump enabling states J_ℓ in our setting. In contrast to the method described here, [MF14] do not compute the shadow of the states where U is entered explicitly. Instead, the complement $\overline{U}$ is added to the invariant, which ensures that U is never entered and the shadow is not included in the time successors in the first place. The entry points of U are then added separately and are constructed as the boundary between the time successors (with invariant $\overline{U}$) and U. Some additional care has to be taken to include only those points on the boundary that are reachable from $(X_0 \oplus F) \cap \overline{U}$ while letting time elapse. In our setting the obtained set is the same as $\mathit{entry}(J_\ell)$.

To compute the time successors with the non-convex invariant $\mathit{Inv}(\ell) \cap \overline{U}$, [MF14] iterate over all convex components P_i of $\mathit{Inv}(\ell) \cap \overline{U}$ and compute the time successors that lie in that component as well as potential entry points to other components P_j, for which the time successors are computed in the next iteration. This process is repeated until no new entry points are obtained.

The main advantage of the algorithm in [MF14] is that it can also be applied to *rectangular automata*, or non time deterministic LHA I, where the dynamics for a variable x are defined as an interval of possible derivatives. In our approach, the exclusion of the shadow doesn't work with rectangular dynamics, because states that lie in the shadow of J_ℓ may still be reachable, which is shown in Figure 3.6. An advantage of the presented method is that we do not have to compute time successors multiple times if the complement $\overline{J_\ell}$ is not convex, whereas if we used the algorithm in [MF14], time successor computation is performed for each component of $\overline{J_\ell}$.

Chapter 4

CEGAR

In this chapter we describe an approach for analyzing urgent LHA using the Counterexample-guided abstraction refinement [CGJ$^+$00] technique. We do this by integrating it into the flowpipe-construction analysis algorithm described in Section 3.3. We first describe the CEGAR technique in a general setting in Section 4.1 where we also highlight related applications to hybrid systems. In Section 4.2 we discuss why applying CEGAR to the analysis of urgent LHA in particular can be an improvement and describe how we integrate refinement into the analysis algorithm. Here, we also lay the formal foundation for Section 4.3, where we focus solely on the counterexample analysis and refinement step. Finally, in Section 4.4 we present optional improvements and heuristics for refinement.

4.1 The CEGAR Technique

Counterexample-guided abstraction refinement, or CEGAR, is an analysis technique for model checking. It was first proposed in [CGJ$^+$00] and has since then applied to various problems. We first outline the general idea and then highlight some applications to hybrid systems related to our approach.

The first step of CEGAR is to create an *abstraction* of the original model. Here, an abstraction describes a simplified model that keeps the properties of interest of the original model. For safety analysis this means that if the abstraction is safe then so is the original model. The abstraction is then analyzed instead of the original, more complex model. If analysis of the abstraction indicates that the system is unsafe, a *counterexample* is constructed. A counterexample is a concrete execution run that can be mapped to a run in the original model. In the context of hybrid systems this is often a path to an unsafe state. The next step is to *analyze* the counterexample, which means checking whether the counterexample can exist in the original model in which case we say that it is *real*. In case the counterexample is real then CEGAR indicates failure, i.e., the analysis property is not satisfied by the model. Otherwise, if the counterexample only exists in the current abstraction, we say that it is *spurious*. In that case the abstraction is *refined*, i.e., it is made more precise by adding more properties of the original model. Counterexample guidance means that refinement is done in a way such that the counterexample does not exist anymore in the refined abstraction. After refinement the new abstraction is analyzed again until either no counterexamples are found or a real counterexample is found. The analysis loop is depicted in Figure 4.1.

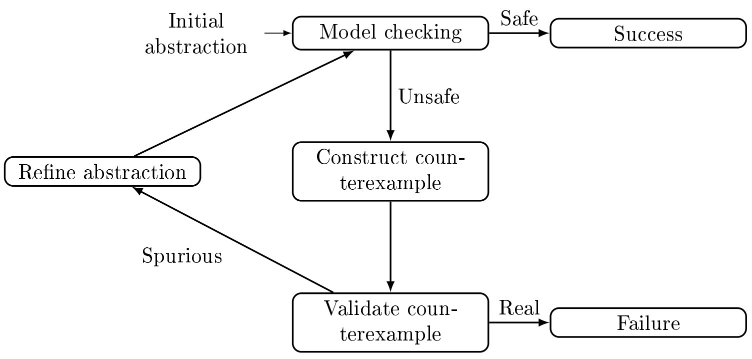


Figure 4.1: General CEGAR loop.

Related approaches. The CEGAR technique has been applied to the reachability problem in the past in different ways. We briefly sketch the main ideas of some approaches and in particular the abstraction and refinement process.

In [CFH$^+$03] the abstraction consists of using different approximations of reachable states. The initial abstraction for example eliminates all continuous variables and thus only considers the discrete part of the analyzed automaton. If a counterexample, i.e., a run to a bad state is found, other approximations of that path are computed. For refinement the abstract state space is then split into states that are reachable in all approximations and those that are not, which allows to refute the reachability of states. This is improved in [FCJK05] to only consider fragments of the counterexample which are partial runs of shorter length. A related approach is presented in [ADI03] where the abstraction consists of dividing the state space into regions by a set of predicates. Refinement then adds more predicates to this set which means that a more precise approximation can be computed.

Somewhat similar to the initial abstraction of [CFH$^+$03], the authors in [JKWC07] use an abstraction for LHA that eliminates a set of variables from the constraints of the automaton, such as invariants, flows or guards. Refinement makes the set of ignored variables smaller, which means that the abstraction gets more precise. This refinement is counterexample guided by using linear programming to identify a minimal set of variables and constraints to refute the current counterexample.

The CEGAR approach in [DKL07] focuses on PLC programs that are modeled by timed automata and uses a similar abstraction refinement by ignoring the values of a shrinking set of abstracted variables. A related method is presented in [NÁW15], where PLC programs with dynamic plant behavior are considered. In their initial abstraction all variables are assumed to have chaotic behavior and in the refinement step concrete dynamics for a subset of variables is added. A notable feature of this approach is that refinement is done on the fly and analysis is not restarted from scratch after a counterexample is refuted. This is achieved by building a reachtree during analysis and removing subtrees that are affected by refinement while leaving the rest of the reachtree unchanged.

In [SÁ18] the *partial-path-refinement* method is presented which, as the name suggests, only refines specific paths in the reachtree. If an unsafe path is found, it is iteratively analyzed with increasingly precise analysis settings such as a smaller time step size or smaller clustering size.

In this work we will use ideas similar to [NÁW15] and [SÁ18] to refine along paths in a search tree without restarting analysis.

4.2 CEGAR for Urgent Automata

We start this section by motivating why we want to use a CEGAR approach for analysis of urgent LHA. One of the main issues when analyzing urgent LHA (see Section 3.3) is that the set difference of two convex sets is not generally convex and thus creates multiple fragments for each segment. For each of these fragments the time successors have to be computed again which can cause an exponential blowup in the number of fragments and thus computation time. To circumvent this we make use of the fact that it may not be necessary to respect urgency of all transitions, i.e., urgency of some transitions can be ignored. We illustrate this situation on an example.

Example 4.2.1. *Recall the vehicle automaton from Example 2.1.1. The vehicle can be either driving or braking and can brake at any time. Here, we consider a modified version in which the vehicle brakes in specific regions. These can for example be regions in which the driver can see speed limit signs which causes them to slow down. This can be modeled by an urgent transition for each region X_{brake} with guard $x \in X_{brake}$. Assume we want to analyze safety of this model with a set of bad states $Bad = \{(\ell, (x, v, a)) \mid x \geq x_{limit} \wedge v \geq v_{max}\}$, which means that after driving some distance x_{limit} the vehicle must not go above the speed v_{max}. Then it may not be necessary to respect every braking region to satisfy the safety condition, since e.g., braking once is enough. In that case the urgency of all other transitions can be ignored.*

The benefit of ignoring urgency is that no set difference computation is necessary and therefore analysis is potentially faster. At the same time however this leads to additional over-approximation of the analyzed automaton, which may not be sufficient to prove safety. We therefore want to apply CEGAR to the analysis of urgent LHA, where the abstraction essentially consists of removing urgent transitions and refinement adds back more and more urgent transitions until safety can be verified. Note that such an abstraction is indeed an over-approximation, i.e., if H is an urgent LHA and H' is a copy of H with a smaller set Urg' of urgent transitions then any run in H is also a run in H', because urgency only impedes time elapse and thus makes the set of reachable states smaller. In particular, if H' is safe then so is H. Note also that since the reachability problem is undecidable we can again only get a decisive answer if the model is safe and the analysis is precise enough to prove it. The goal of our CEGAR analysis is therefore to return the same safety result as reachability analysis applied to the original automaton would. This also makes the presented algorithm somewhat independent of the used reachability algorithm, although it is aimed at the analysis method in Section 3.3. As subject of future work however the ideas can be transferred to e.g., the analysis method in Section 3.4, resulting in a CEGAR algorithm for urgent time deterministic LHA I.

As observed in [NÁW15], a major drawback of applying CEGAR in a straightforward manner to the reachability problem is that after each refinement step analysis is restarted and the entire reachtree has to be computed again. Parts of the reachtree may however be unaffected by the refinement step, or refinement may not be necessary to verify safety on some paths. The goal is therefore to only refine along unsafe paths in the tree, while leaving the rest unchanged. In [NÁW15] this is achieved by deleting and recomputing subtrees that are affected by the refinement step. Another approach is presented in [SÁ18], where the nodes have multiple levels, each corresponding to a refinement level of the model on the given path. We will explore a method to refine paths in Section 4.3.

To formalize path refinement we first extend reachtrees to store refinement information, i.e., which urgent transitions were used to compute successor nodes. Here we also extend analysis paths from Definition 2.3.3 to store similar information. The plan is then to use analysis paths to describe counterexamples as runs leading to a bad state and to refine the model along these unsafe paths.

Refinement trees. We extend Definition 2.3.2 to abstractions of urgent automata:

Definition 4.2.1. For a hybrid automaton $H = (\mathit{Loc}, \mathit{Var}, \mathit{Flow}, \mathit{Inv}, \mathit{Edge}, \mathit{Urg}, \mathit{Init})$ a *refinement reachtree* is a tuple

$$T_H = (\mathit{Nodes}, \mathit{Root}, \mathit{Succ}, \mathit{State}, \mathit{Trace}, \mathit{Refined}),$$

such that $(\mathit{Nodes}, \mathit{Root}, \mathit{Succ}, \mathit{State}, \mathit{Trace})$ is a reachtree of H and

$$\mathit{Refined}\colon \mathit{Nodes} \to 2^{\mathit{Urg}}$$

is a function such that $\mathit{Refined}(N) \subseteq \mathit{Urg}_\ell$ for all $N \in \mathit{Nodes}$ where $\mathit{State}(N) = (\ell, S)$ and Urg_ℓ is the set of outgoing urgent transitions of ℓ. We call a transition $t \in \mathit{Urg}_\ell$ *refined in* N if $t \in \mathit{Refined}(N)$ and otherwise *unrefined in* N. A node N' *refines* another node N if $\mathit{State}(N') = \mathit{State}(N)$ and $\mathit{Refined}(N') \supseteq \mathit{Refined}(N)$.

Semantically, we treat unrefined transitions the same as non-urgent transitions, which means that they do not impede time elapse. In particular, we assume that the operators $\mathrm{FP}(N)$ and $\mathrm{JS}(N)$, which compute the flowpipe and jump successors of the initial state set of N, are modified accordingly. Analogously to reachtrees we call a node N *complete* if the successor nodes contain all jump successors, i.e., if $\mathrm{JS}(N) = \{\mathit{State}(N') \mid (N, N') \in \mathit{Succ}\}$. Next we extend analysis paths to include the refinement of transitions:

Definition 4.2.2. An *analysis path* p is a tuple

$$p = (U_0, \tau_0, e_0, U_1, \tau_1, e_1, \ldots, U_{n-1}, \tau_{n-1}, e_{n-1}, U_n),$$

where $n \geq 0$ and $\tau_i \in \mathbb{I}_\mathbb{R}$, $e_i \in \mathit{Edge}$ for $i = 0, \ldots, n-1$ and $U_i \subseteq \mathit{Urg}$ for all $i = 0, \ldots, n$. We write $|p| := n$ for the *length* of p. We denote the components of p by $U_i(p), e_i(p)$ and $\tau_i(p)$.

For an analysis path $p = (U_0, \tau_0, e_0, U_1, \tau_1, e_1, \ldots, U_{n-1}, \tau_{n-1}, e_{n-1}, U_n)$, the set U_i is the set of refined or urgent transitions at depth i. The tuple (τ_i, e_i) is the trace between the nodes at depth i and $i+1$. A node N in a refinement tree induces an analysis path $p(N)$, which encodes the computations that were done to construct N. Similarly, if R is an ancestor of N then $p(R, N)$ is the analysis path obtained by following the edges from R to N. For examples of the construction of $p(N)$ and $p(R, N)$, see Example 4.2.2.

We now formalize refinement of paths as a partial order on the set of all analysis paths. Intuitively, paths are *similar* if they encode the same jumps and jump-timings and a path refines another path if it has more refined transitions.

Definition 4.2.3. Let p, p' be analysis paths. We say that p and p' are *similar* and write $p \sim p'$ if $|p| = |p'|$ and for all $i = 0, \ldots, |p| - 1$ it holds that $e_i(p) = e_i(p')$ and $\tau_i(p) = \tau_i(p')$. If additionally $U_i(p) \subseteq U_i(p')$ for all $i = 0, \ldots, |p|$ then p' *refines* p and we write $p' \sqsubseteq p$.

Finally, we define the set of nodes that belong to an analysis path in a given refinement tree T_H, in order to define the nodes that must be completed to refute a counterexample. Here we assume that the path can start at some node $R \in \mathit{Nodes}$, which will be useful when reasoning about partial paths, i.e., paths that do not start in the root node. The nodes of an analysis path p then include R and all descendant nodes with the trace induced by the components of p. Additionally, the nodes must have the refined transitions as they are given in p. Formally, we define the nodes of p inductively:

Definition 4.2.4. Let p be an analysis path, T_H a refinement tree and $R \in \mathit{Nodes}$ a node of T_H with $\mathit{Refined}(R) = U_0(p)$. We define the set $\mathrm{nodes}_R(p)$ of nodes belonging to p starting in R:

- If $p = (U_0)$ is of length 0 then $\mathrm{nodes}_R(p) := \{R\}$
- If $|p| \geq 1$ then

$$\begin{aligned}\mathrm{nodes}_R(p) := \mathrm{nodes}_R(p') \cup \{N' \mid \exists N \in \mathrm{nodes}(p'_R).\, &\mathrm{d}(N) = \mathrm{d}(R) + n - 1 \wedge \\ &(N, N') \in \mathit{Succ} \wedge \mathit{Trace}(N, N') = (\tau, e) \wedge \\ &\tau \cap \tau_{n-1} \neq \emptyset \wedge e = e_{n-1} \wedge \\ &\mathit{Refined}(N') = U_n(p)\},\end{aligned}$$

  where $p' = (U_0, \tau_0(p), e_0(p), \ldots, U_{n-1}(p))$ and $\mathrm{d}(N)$ is the depth of N.

For brevity we sometimes omit R and mean $\mathrm{nodes}(p) = \mathrm{nodes}_{\mathit{Root}}(p)$ in that case, which means that paths start in the root node by default.

An analysis path p is *complete* in T_H if every node of p is complete in T_H. The reachable states of p is the union of all flowpipes of the nodes of p, i.e.,

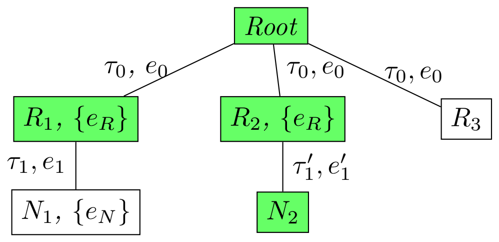
$$\mathrm{Reach}(p) = \bigcup_{\substack{N \in \mathrm{nodes}(p) \\ \Omega \in \mathrm{FP}(N)}} \Omega.$$

Since refinement makes analysis more precise we have the following useful lemma.

Lemma 4.2.1. *Assume that $p' \sqsubseteq p$ and that p and p' are complete in T_H. Then* $\mathrm{Reach}(p') \subseteq \mathrm{Reach}(p)$.

We illustrate some of the concepts defined so far at the hand of an example refinement tree.

Example 4.2.2. *Consider the following refinement tree with node set $\mathit{Nodes} = \{\mathit{Root}, R_1, R_2, R_3, N_1, N_2\}$. We color the completed nodes Root, R_1, R_2 and N_2 green and the unexplored nodes R_3 and N_1 white. The edges between nodes are labeled with the corresponding Trace value and the nodes are annotated with their set of refined nodes, which means that $\mathit{Refined}(N_1) = \{e_N\}$ and $\mathit{Refined}(R_1) = \mathit{Refined}(R_2) = \{e_R\}$. If all transitions are unrefined we omit this annotation, for example we have $\mathit{Refined}(N) = \emptyset$. Similarly, we omit the initial states of the nodes as they are usually not relevant to us.*



We now consider the example paths

$$p_1 = (\emptyset, \tau_0, e_0, \{e_R\}, \tau_1, e_1, \{e_N\}),$$
$$p_2 = (\emptyset, \tau_0, e_0, \{e_R\}, \tau_1, e_1, \emptyset).$$

The set of nodes of p_1 is $\text{nodes}(p_1) = \{Root, R_1, R_2, N_1\}$*. Note that R_3 does not belong to p_1 because e_R is not refined in R_3. Since N_1 is not complete and belongs to p_1, it is not complete. If we instead consider the path p_2 then N_1 does not belong to p_2 anymore, so p_2 is complete.*

The path induced by N_2 is

$$p(N_2) = (\emptyset, \tau_0, e_0, \{e_R\}, \tau_1', e_1', \emptyset).$$

Note that R_1 also belongs to $p(N_2)$. Finally, the path from R_2 to N_2 is given by $p(R_2, N_2) = (\{e_R\}, \tau_1', e_1', \emptyset)$.

Integration of CEGAR. We now explain how the CEGAR loop depicted in Figure 4.1 is integrated into the analysis algorithm. For that we assume that reachability analysis indicates potential unsafety of the abstracted automaton and the goal is to construct a counterexample and refine the model if it is spurious. A *counterexample* for safety can be considered as a run from an initial state to an unsafe state. Since our analysis method is over-approximative we here define a counterexample as an analysis path from the root to an unsafe node, i.e., a node whose segments contain bad states. Alternatively, we also call the unsafe node itself a counterexample and mean the path induced by the node. If an unsafe node is found, the analysis path can then easily be constructed from the refinement tree we build during analysis. We call a counterexample path p *spurious* if there is a complete analysis path $p' \sim p$ such that p' is safe, i.e., contains only safe nodes. Note that by Lemma 4.2.1, spurious counterexamples cannot exist in the original, completely refined automaton and therefore the automaton is safe if a complete refinement tree contains only spurious counterexamples. If the counterexample is not spurious we say that it is *real*, which means that it cannot be refuted by further refinement, so safety cannot be verified. However, this does not mean that there must exist a concrete unsafe run in the original automaton, since our analysis method is still over-approximative.

After constructing a counterexample, the next step is to validate or refute it and if it spurious, refine the abstraction by adding in more refined urgent transitions. Instead of doing counterexample validation and refinement as separate steps we consider them as interleaved steps, as proposed in [CFH$^+$03]. A counterexample path is therefore refuted by refining the automaton along the path and checking if a bad state is still potentially reachable. If further refinement is not possible, or cannot improve precision further, the counterexample cannot be refuted and the algorithm indicates failure.

The analysis algorithm is shown in Algorithm 4, where analysis is executed normally until an unsafe node is found. In that case, the *refine* step attempts to refute the counterexample by implicitly refining the given analysis path. If the counterexample can be refuted, it will also push successor nodes of the refined path to the global queue Q, so that analysis can continue.

Algorithm 4: CEGAR Reachability algorithm

```
Input  : Urgent LHA $H$ with initial state set $(\ell_0, Init)$ and bad states $Bad$.
Output: Safety of $H$.
$Q := \{\text{node}(\ell_0, Init, \emptyset, Null)\}$ ;                  // global queue
while $Q \neq \emptyset$ do
    $N := \text{pop}(Q)$;
    if $!\,\text{Safe}(\text{FP}(N))$ then
        $success = refine(N)$ ;                                 // modifies $Q$
        if $!success$ then
            return $UNSAFE$;
    else
        for $successorState \in \text{JS}(N)$ do
            $\text{push}(Q, \text{node}(successorState, \emptyset, N))$ ; // unrefined children
        end
end
return $SAFE$;
```

4.3 Path Refinement

In this section we discuss how the *refine* step in Algorithm 4 works in detail. Throughout this section we fix a partially explored refinement tree T_H and an unsafe node N with analysis path $p = p(N)$.

Let $p' \sim p$ be a similar analysis path which we want to complete in the refinement tree in order to refute the counterexample. The choice of p' is a matter of *refinement strategy* in that multiple options are possible to construct it. Here, we fix a strategy S which maps an analysis path p to a similar path $S(p) \sim p$. We require that repeated application of S eventually reaches a fixpoint in order to guarantee termination of the algorithm. Ideally, S is also counterexample guided in order to improve performance. The plan is to analyze $S(p)$, i.e., complete it in the refinement tree. If this proves safety of $S(p)$ then p is spurious and we are done and otherwise we apply S iteratively until safety can be verified or S reaches a fixpoint in which case the counterexample is declared real.

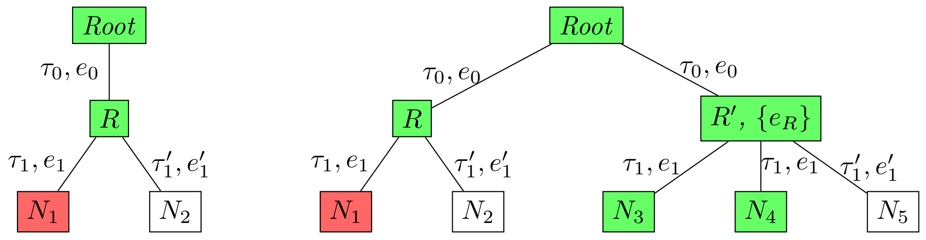
A simple example strategy chooses some unrefined transition $e \notin U_k(p)$ for some k and returns the analysis path $p' \sim p$ with $U_k(p') = U_k(p) \cup \{e\}$ and $U_i(p') = U_i(p)$ for $i \neq k$. If all transitions are refined, the strategy returns p, so that termination is guaranteed. We will explore a counterexample guided refinement strategy later, but first focus on how the completion of a chosen $p' \neq p$ is integrated into the refinement tree.

Now let k be the smallest index such that $U_k(p') \neq U_k(p)$ and let R be the unique ancestor of N with depth k. The first step is to create a node R' with $Refined(R') = U_k(p')$. Here, we make the decision to create R' as a sibling node of R. We could also attempt to replace R with R', delete the subtree rooted at R and recompute it. This is a similar idea to [NÁW15], however we can only do this if R' has strictly more refined transitions than R. Another possible disadvantage is that the deleted subtree could already be large and only contain a single unsafe path which results in redundant computation steps. Other options include using multi-leveled tree nodes similar to what is done in the partial path refinement algorithm in [SÁ18], so that

R and R' are summarized in one node as different refinement levels. The problem with this approach is that in contrast to [SÁ18], our refinement levels are not totally ordered, since it is not clear which set of refined transitions gives more precise analysis results. Additionally, refinement levels are different for each node, since their locations have different outgoing urgent transitions. Finally, another idea suggested in [SÁ18] is to create a new refinement tree for each refinement iteration. This adds additional challenge in keeping track of the individual trees and also creates redundancy since the ancestors of R and R' are identical in both trees. We therefore only build the subtree starting at R' and integrate it into the existing tree by making R' a sibling of R. This makes it easy to keep track of and reuse refinement levels, i.e., variations of the set of urgent transitions, since we know that they are siblings of each other. A technicality that needs to be considered in practice is what happens when $R = Root$. In that case we create a separate refinement tree, with root R', i.e., we consider the roots of these multiple refinement trees as siblings.

After R' has been created, note that it is sufficient to complete the tail of p' starting at R'. This is because any nodes of depth less or equal than R' belong to p and will be completed in the outer analysis loop, i.e., outside of the current refinement iteration. If any of those nodes are also unsafe, an additional refinement iteration will be triggered for them. In order to reduce redundancy we thus consider only the tail of p' and complete it as subtree of R'. For that we use a similar algorithm as the path analysis in Algorithm 2, with the only adaption that transitions on the path are refined accordingly to p'. In particular, we use a second queue Q_r which is local to the refinement algorithm. If a new unsafe node is found, the refinement strategy is applied again until it returns a fixpoint, in which case the counterexample is real. Finally, successors of nodes that lie at the end of the path need to be completed as well, but they do not belong to p' anymore. They are therefore pushed to the global queue Q after the counterexample is successfully refuted.

Example 4.3.1. *We give an example of a successful refinement iteration. The initial situation is depicted in the left refinement tree below. The node N_1 is unsafe, indicated by the red coloring. The counterexample path is $p(N_1) = (\emptyset, \tau_0, e_0, \emptyset, \tau_1, e_1, \emptyset)$. For refinement we choose the path $p' = (\emptyset, \tau_0, e_0, \{e_R\}, \tau_1, e_1, \emptyset)$. Thus the node R' with refined transition e_R is created as sibling of R. The refinement tree after path refinement is shown on the right hand side. The nodes R', N_3 and N_4 are completed because they lie on p'.*



Note that R' has three children while R only has two. This can for example happen because the set difference computation creates more fragments and thus more jump successors. Note also that the node N_5 is created, so that R' is complete but not explored because it does not belong to p'.

The refinement algorithm is summarized in Algorithm 5. It keeps a local queue Q_r to hold nodes that still need to be completed. Additionally, we use a list *endOfPath* to remember nodes that lie at the end of the final analysis path. Lines 1 to 8 are the initial refinement step where the first new node R' is created and pushed to Q_r. In lines 10 to 28 the analysis path starting at R' is analyzed and if new unsafe nodes are found, more refinement iterations are triggered in lines 14 to 19. Finally, in lines 30 and 31 the unrefined path successors are pushed to the global queue so that analysis can continue.

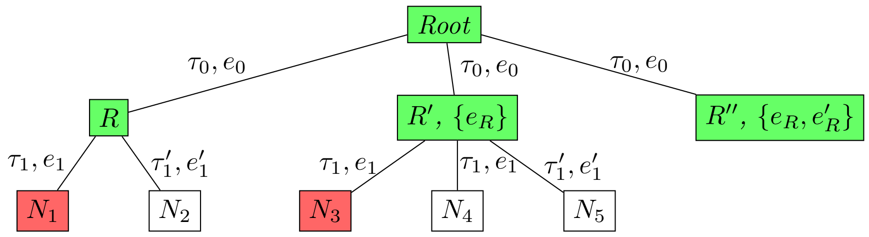
Algorithm 5: refine

```
   Input  : Unsafe node $N$ and refinement strategy $S$.
   Output: Indicate whether $N$ could be refuted.
 1 $p = path(N)$;
 2 $p' = S(p)$;
 3 $k = \min(\{k \mid U_k(p') \neq U_k(p)\})$;
 4 if $k == \infty$ then
 5     return false ;                                   // fixpoint reached
 6 end
 7 $R = ancestor(N, k)$ ;                              // ancestor of depth $k$
 8 $Q_r := \{\text{node}(\ell_R, Init_R, U_k(p'), parent(R))\}$;
 9 $endOfPath = \emptyset$;
10 while $Q_r \neq \emptyset$ do
11     $p = p'$;
12     $M := \text{pop}(Q_r)$;
13     if $!\,\text{Safe}(\text{FP}(M))$ then
14         $p' = S(p)$;
15         $k = \min(\{k \mid U_k(p') \neq U_k(p)\})$;
16         if $k == \infty$ then
17             return false;
18         end
19         $\text{push}(Q_r, \text{node}(\ell_R, Init_R, U_k(p'), parent(R)))$;
20     end
21     else
22         if $\text{depth}(M) == |p|$ then
23             $\text{push}(endOfPath, M)$;
24         end
25         else
26             $createAndPushChildrenOnPath(Q_r, N, p')$;
27         end
28     end
29 end
30 for $M \in endOfPath$ do
31     $createAndPushChildren(Q, M)$ ;                  // unrefined children
32 end
33 return true;
```

Avoiding redundancy. While Algorithm 5 correctly analyzes counterexamples it can do some redundant computation steps. We cover two sources of redundancy and explain how they can be minimized.

The first source of redundancy occurs if a node M is in the queue Q_r but an ancestor R_M of M was the refinement node, in a previous step. With *refinement node* we here refer to the node of depth k for which a sibling is created in lines 8 and 19 of Algorithm 5. In this case M does not need to be completed anymore, because the path will also be completed below the refined node R'_M. We illustrate this on an example:

Example 4.3.2. *We continue Example 4.3.1 but assume that we just finished completing R'. The current refinement queue is $Q_r = \{N_3, N_4\}$. Assume here that N_3 turns out to be unsafe, so that another refinement iteration is triggered. Depending on the order that the nodes are explored, this may result in the following refinement tree.*



Here, refinement of e'_R had the effect that R'' has no children, so the refined path is immediately complete. In the end N_4 is still unexplored and in the queue Q_r but the original counterexample is already refuted, so we can ignore N_4.

We therefore keep an additional list *refined* which holds all the refinement nodes. Whenever a refinement iteration is triggered in Algorithm 5, lines 14 to 19, R is added to this list. For every node M that is taken from the queue, we first check whether an ancestor of M is in that list and if so we can skip analysis of M. Similarly, only nodes in *endOfPath* that do not have any ancestors in *refined* are pushed to Q in lines 30 and 31.

The second source of redundancy is that often we choose the same refinement nodes over and over again. To avoid multiple computations of the same path analysis the algorithm therefore checks whether the refined node R' already exists. This is easy to do because R' must be a sibling of R if it exists. If R' exists it can also happen that successor nodes that lie on the path p' have already been explored in previous refinement iterations. The algorithm therefore follows these nodes as long as possible and only computes successors for unexplored nodes.

Refinement strategy. We now explain the counterexample guided refinement strategy S we use in Algorithm 5. The first step is to find a node R and an unrefined transition e in R such that the pair (R, e) is *suitable for refinement*, a condition we will define in a moment. If no such pair exists, p is returned as fixpoint which indicates that further refinement is pointless. Otherwise, we return the path $S(p) = p' \sim p$ with refined transitions

$$U_i(p') = \begin{cases} U_i(p) & \text{if } i < k \\ U_i(p) \cup \{e\} & \text{if } i = k \\ \emptyset & \text{if } i > k. \end{cases} \tag{4.1}$$

We first define a necessary condition that the pair (R, e) must satisfy in which case we say that it is suitable for refinement. After that we explain why we define p' this way.

Let R be an ancestor of N and $e = (\ell, g, r, \ell')$ be unrefined in R. Since N is unsafe there are segments of $\mathrm{FP}(R)$ and subsets of those segments that lead to a bad state in N when following the path p. For a segment Ω we call this set of predecessors Ω_{bad}. Assume that for all segments Ω we have that $\Omega_{bad} \cap J_e = \emptyset$, where J_e is the transition enabling set

$$J_e = g \cap r^{-1}(Inv(\ell')),$$

that is removed from Ω when refining e (see Equation (3.5)). In that case refining e in R cannot help because the same predecessors Ω_{bad} will lead to bad states again. A necessary condition is therefore that

$$\Omega_{bad} \cap J_e \neq \emptyset, \tag{4.2}$$

for some segment $\Omega \in \mathrm{FP}(R)$. Note that this condition is not sufficient, i.e., even if it is satisfied refining R with e may not be enough to verify safety.

To check whether Equation (4.2) holds for some segment Ω we compute the set $\Omega \cap J_e$ and complete the partial path $p(R, N)$ from R to N, starting from a fresh node with initial set $\Omega \cap J_e$. If this is safe for every segment then Equation (4.2) cannot hold and thus the pair (R, e) is not suitable for refinement.

An algorithm to find a suitable refinement node is given in Algorithm 6.

Algorithm 6: findRefinementNode

Input : Unsafe node N.
Output: A pair (R, e) suitable for refinement.

```
1  for $(R, e) \in$ ancestors$(N) \times Urg$ do
2      if $e \notin Refined(R)$ then
3          for $\Omega \in \mathrm{FP}(R)$ do
4              $R_e :=$ node$(\ell_R, J_e \cap \Omega, \emptyset, Null)$;
5              if isUnsafe(completePath$(R_e, p(R, N))$) then
6                  return $(R, e)$;
7          end
8      end
9  end
10 return $(Null, Null)$
```

It works in a straightforward manner by iterating over all pairs (R, e) and the segments $\Omega \in \mathrm{FP}(R)$ and checking whether $p(R, N)$ is safe when starting from $\Omega \cap J_e$. The path analysis is done using Algorithm 2. To order the pairs (R, e) we make the choice of starting at the root node *Root* and increasing in depth until N is reached. The ordering of the transitions in each node is left arbitrary for now. Another option would be to go in reverse order starting with N, with the advantage that shorter paths $p(R, N)$ are considered first. On the other hand it is also reasonable to refine nodes closest to the root as soon as possible. This is because if multiple transitions have to be refined to refute the counterexample then a sufficient analysis path is found in fewer iterations than if nodes closest to N are refined first, which we illustrate in Example 4.3.3. In Section 4.4 we will present some more advanced heuristics for ordering the pairs (R, e).

Example 4.3.3. *Assume that N is a counterexample with path $p = (\tau_1, e_1, \emptyset, \tau_2, e_2, \emptyset)$. Assume further that to refute N we need the path $p' = (\tau_1, e_1, \{e_1\}, \tau_2, e_2, \{e_2\})$, i.e., refining only e_1 or e_2 is not sufficient. Then both e_1 and e_2 are suitable for refinement. If e_2 is refined first, then the paths*

$$(\tau_1, e_1, \emptyset, \tau_2, e_2, \{e_2\}),\ (\tau_1, e_1, \{e_1\}, \tau_2, e_2, \emptyset),\ (\tau_1, e_1, \{e_1\}, \tau_2, e_2, \{e_2\}),$$

are attempted until p' is found. If on the other hand e_1 is refined first then only two refinement iterations

$$(\tau_1, e_1, \{e_1\}, \tau_2, e_2, \emptyset),\ (\tau_1, e_1, \{e_1\}, \tau_2, e_2, \{e_2\}),$$

are necessary.

After finding a suitable pair (R, e) our strategy constructs the path p' as in Equation (4.1). Here, we made the decision of setting $U_i(p') = \emptyset$ for $i > k$, even though $U_i(p)$ may be non-empty, which means that all descendants of R' are again initially unrefined. We do this because transition refinement can change the tree structure, as in Example 4.3.1, so that it may not be necessary to include all refinements of p. Our strategy therefore aims at doing as few unnecessary refinements as possible, and thus transitions are only refined on demand.

4.4 Improvements

This section covers some optional variations of the refinement algorithm presented in Section 4.3 that aim at improving the performance. In Chapter 5 we will then benchmark these variations and compare them with the base version to see which ones are actually an improvement.

Pruning segments and refining halfspaces. One simple possible improvement that we can make is *pruning* segments. Here, we check for each computed segment whether it is completely contained in the jump enabling set J_e of an urgent jump e. In that case, set difference computation with J_e would yield the empty set and in particular would not create multiple fragments. Thus we can simply drop the segment and stop time successor computation in that case, regardless of whether e is refined in the current node or not.

Similarly, it can be reasonable to always refine transitions for which J_e is a halfspace. In that case, set difference of a segment with J_e results in a single fragment, which means that the reasoning for using CEGAR does not apply to e. This can be achieved by adding all such transitions to the set $\mathit{Refined}(N)$ for all nodes N upon creation.

Heuristics. We now consider some heuristics for ordering the refinement candidates in Algorithm 6. Recall that by default we order the nodes by depth in ascending order and assume an arbitrary ordering of the urgent transitions in each node. Here, we present three heuristics, which we call the *count*, *volume* and *constraint-count* heuristics.

The *count* heuristic keeps a counter for each transition and orders the pairs (R, e) by the counter values for e in descending order. Each time the pair is found suitable for refinement, the counter for the chosen transition is increased by one. To break ties between nodes, we fall back to the default ordering and take the node with lower depth. A motivation for the count heuristic is that candidates that were often suitable for refinement may be more likely to work in the future as well. Additionally, we can potentially save time because it is more likely that nodes on the refinement path are already complete.

An alternative implementation of the count heuristic is to only increase the counter for the *last* transition that was refined before the counterexample is refuted. The reasoning here is that the refinement candidate was not only suitable for refinement but actually helped in refuting a counterexample. In practice we observed that both heuristics lead to similar results so we only consider the first variant.

Next we consider the *volume* heuristic. Here, we want to compute the sum

$$\sum_{\Omega \in \mathrm{FP}(R)} \mathrm{volume}(\Omega \cap J_e), \tag{4.3}$$

and order the candidates in descending order. A problem is that the sum in Equation (4.3) may have to be computed for many candidates. Since the exact computation of the volume of arbitrary polytopes is likely too much overhead, we approximate it roughly by taking the volume of the *bounding box* of $\Omega \cap J_e$. The bounding box here is the smallest box containing $\Omega \cap J_e$. The volume of the box can then easily be computed as the product of the volume of the individual intervals. The idea behind the volume heuristic is that a large intersection with J_e means that set difference computation with J_e will exclude a large set. It may therefore be more likely that refinement excludes more predecessors of bad states, so that refinement is more helpful.

Finally, we present the *constraint-count* heuristic. This heuristic is rather static, especially compared to the volume heuristic. The idea is to order the candidates (R, e) by the number of halfspaces defining J_e in ascending order. Ties between nodes are again broken by taking the node with the lower depth. If one node has multiple transitions with the same number of halfspaces we again assume an arbitrary ordering of these transitions. The reason it may be useful to consider transitions with low constraint-count value first is not that they are more likely to be suitable for refinement but that refinement may be easier. This is because set difference computation in the worst case creates one fragment for each halfspace of J_e, as explained in Section 3.2. Therefore, refining transitions where the jump enabling set has few halfspaces can result in less fragmentation and thus faster successor computation.

Refinement levels. We now introduce a new *refinement level* for urgent transitions. Let R be a node in a refinement tree and e an urgent transition. So far, e can be either unrefined in R, which means that it is treated as a non-urgent transition, or it is refined. This can also be considered as two refinement levels and we want to introduce a third one that is a middle ground between the two. The aim is to adapt the time successor computation by replacing each segment Ω with a convex set Ω' such that $\Omega \setminus J_e \subseteq \Omega' \subseteq \Omega$. Since we compute a single convex set, this does not have the fragmentation problem of set difference, but makes the analysis more precise than taking the entire segment. By definition, the smallest convex set we can obtain for Ω' is the convex hull of $\Omega \setminus J_e$. A basic approach to compute the convex hull is to first compute the set difference as a collection of polytopes and take the convex union of them. However, if we use $\mathcal{H}$-polytopes to compute the set difference as assumed in Section 3.2, then computation of Ω' involves taking the union of potentially many $\mathcal{H}$-polytopes. This can be costly in higher dimensions when there are many polytopes, as it is usually done by converting each polytope into $\mathcal{V}$-representation, taking their union and converting back to $\mathcal{H}$-polytopes [Sch19].

We here consider an alternative approach for which we only need to compute the vertices of a single $\mathcal{H}$-polytope. It is based on the following lemma.

Lemma 4.4.1. *Let $\Omega = \operatorname{chull}\{v_1, \dots, v_{n+1}\} \subseteq \mathbb{R}^d$ be a polytope and let $J \subseteq \mathbb{R}^d$ be convex such that $v_{n+1} \in J$. Let $\alpha_1, \dots, \alpha_n \in [0,1]$ be maximal such that $u_i \in \Omega \cap J$ for all $i \in \{1, \dots, n\}$, where*

$$u_i := \alpha_i v_i + (1 - \alpha_i) v_{n+1}.$$

We define $\Omega' := \operatorname{chull}\{v_1, \dots, v_n, u_1, \dots, u_n\}$. Then $\Omega \setminus J \subseteq \Omega'$.

Intuitively, the idea is to replace each vertex of Ω that lies in J_e with new vertices that are obtained by maximizing in the direction of the other vertices of Ω, while staying inside $\Omega \cap J_e$. By Lemma 4.4.1 this results in an over-approximation of $\Omega \setminus J_e$ and as we will show later, doing this for all vertices gives the closure of the convex hull. To prove it we require *Farkas' Lemma* [Far02], which is a classic lemma about solvability of systems of linear equations. Different equivalent variations of the lemma exist and we here use a formulation from [Zie95], where a proof can be found as well.

Lemma 4.4.2 (Farkas' Lemma). *Let $A \in \mathbb{R}^{m \times d}$ be a matrix and $z \in \mathbb{R}^m$ a vector. Either there exists a point $x \in \mathbb{R}^d$ with $Ax = z$ and $x \geq 0$, or there exists $y \in \mathbb{R}^m$ such that $A^T y \geq 0$ and $y^T z < 0$.*

Here, we understand inequalities between vectors component-wise. With Farkas' Lemma we are ready to prove Lemma 4.4.1. The main steps of the proof are illustrated in Figure 4.2.

Proof of Lemma 4.4.1. We can assume without loss of generality (after shifting Ω) that v_{n+1} is the origin point. Let $J' := \operatorname{chull}(\{v_{n+1}, u_1, \dots, u_n\})$. We claim that $\Omega' \cup J' = \Omega$. The inclusion $\Omega' \cup J' \subseteq \Omega$ is clear since Ω is convex and contains all vertices of Ω' and J'. For the other inclusion first note that if any $\alpha_i = 0$ then $u_i = v_{n+1}$ and the claim is obvious because Ω' contains all vertices of Ω. We therefore assume that $\alpha_i \neq 0$ for all $i \in \{1, \dots, n\}$. Let $x \in \Omega$ be arbitrary and we want to show that $x \in \Omega'$ or $x \in J'$. Since Ω is the convex hull of $\{v_1, \dots, v_{n+1}\}$ and $v_{n+1} = 0$, there are coefficients $\lambda_1, \dots, \lambda_n \geq 0$ such that $x = \sum_{i=1}^n \lambda_i v_i$ and $\sum_{i=1}^n \lambda_i \leq 1$. Further, because all α_i are nonzero we can write $x = \sum_{i=1}^n \frac{\lambda_i}{\alpha_i} u_i$. Thus if $\sum_{i=1}^n \frac{\lambda_i}{\alpha_i} \leq 1$ then $x \in J'$.

We assume otherwise $x \notin J'$ and that $\sum_{i=1}^n \frac{\lambda_i}{\alpha_i} > 1$. We want to show that $x \in \Omega'$. By definition of Ω', we know that $x \in \Omega'$ if and only if there are coefficients $\mu_1, \dots, \mu_{2n} > 0$ such that $x = \sum_{i=1}^n \mu_i v_i + \sum_{i=1}^n \mu_{n+i} u_i$ and $\sum_{i=1}^{2n} \mu_i = 1$. Substituting $u_i = \alpha_i v_i$ the first equation is equivalent to $x = \sum_{i=1}^n (\mu_i + \alpha_i \mu_{n+1}) v_i$. Thus if we can find $\mu_1, \dots, \mu_{2n} > 0$ such that $\mu_i + \alpha_i \mu_{n+1} = \lambda_i$ for $i \in \{1, \dots, n\}$ and $\sum_{i=1}^{2n} \mu_i = 1$ then $x \in \Omega'$.

We therefore want to show that the system of equations

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 & \alpha_1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 & \alpha_2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots & \vdots & \vdots & \dots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & 0 & 0 & \dots & \alpha_n \\ 1 & 1 & 1 & \dots & 1 & 1 & 1 & 1 & \dots & 1 \end{pmatrix} \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_{2_n} \end{pmatrix} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \\ 1 \end{pmatrix} \tag{4.4}$$

has a solution with $\mu = (\mu_1, \mu_2, \dots, \mu_{2_n})^T \geq 0$. Assume that eq. (4.4) has no such solution. By Farkas' Lemma, there exists $y \in \mathbb{R}^{n+1}$ such that

$$\begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 1 \\ 0 & 1 & 0 & \dots & 0 & 1 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 1 \\ \alpha_1 & 0 & 0 & \dots & 0 & 1 \\ 0 & \alpha_2 & 0 & \dots & 0 & 1 \\ \vdots & \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & \alpha_n & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n+1} \end{pmatrix} \geq 0 \text{ and } \sum_{i=1}^{n} \lambda_i y_i + y_{n+1} < 0. \tag{4.5}$$

There are two cases to consider:

Case 1: $y_{n+1} \geq 0$. Let y_j be minimal, i.e., $y_i \geq y_j$ for all i. Note that $y_j < 0$ because otherwise the second inequality in eq. (4.5) cannot hold. Then $\sum_{i=1}^n \lambda_i y_i + y_{n+1} \geq \left(\sum_{i=1}^n \lambda_i\right) y_j + y_{n+1} \geq y_j + y_{n+1} \geq 0$ which is a contradiction. The last inequality comes from the j-th row in the first inequality of eq. (4.5).

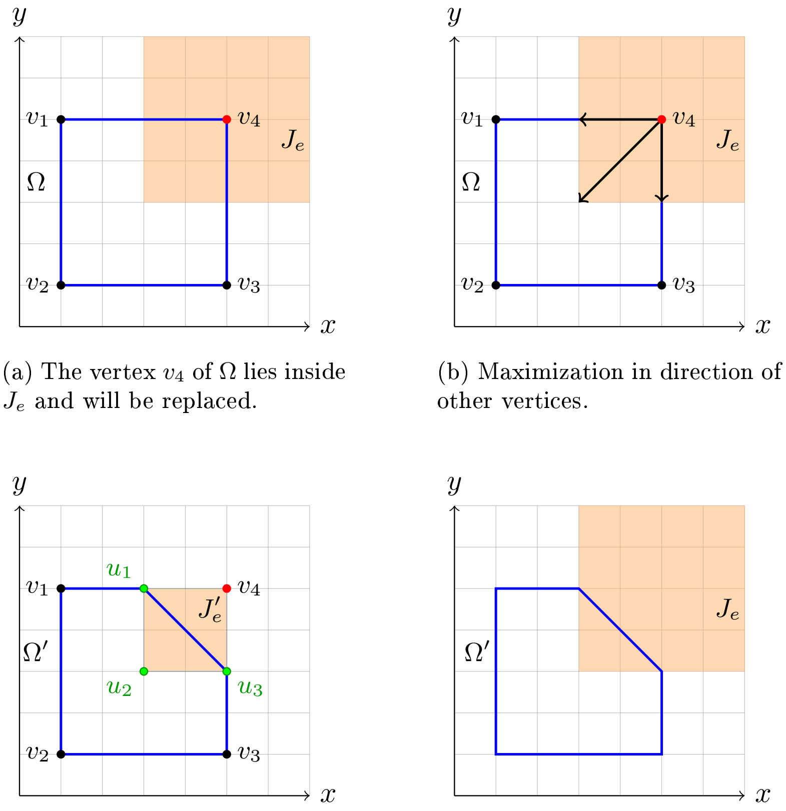
Case 2: $y_{n+1} < 0$. In this case, all other $y_i \geq 0$ for $i \in \{1, \dots, n\}$. Let $j = \operatorname{argmin}_i\{\alpha_i y_i\}$, i.e., $\alpha_j y_j \leq \alpha_i y_i$ for all $i \neq j$. Then

$$\begin{aligned} \sum_{i=1}^{n} \lambda_i y_i + y_{n+1} &= \sum_{i=1}^{n} \alpha_i \frac{\lambda_i}{\alpha_i} y_i + y_{n+1} \\ &\geq \alpha_j y_j \left(\sum_{i=1}^{n} \frac{\lambda_i}{\alpha_i} \right) + y_{n+1} \\ &\geq \alpha_j y_j + y_{n+1} \geq 0, \end{aligned}$$

which is again a contradiction. In the last step we used the earlier assumption $\sum_{i=1}^n \frac{\lambda_i}{\alpha_i} > 1$.

Since both cases lead to a contradiction, the system eq. (4.5) has no solution and therefore the system eq. (4.4) has a solution which means that $x \in \Omega'$ and we have proved the claim that $\Omega = \Omega' \cup J'$.

Finally, since J is convex and contains all vertices of J' we have that $J' \subseteq J$. In conclusion we have $\Omega \setminus J = (\Omega' \cup J') \setminus J = \Omega' \setminus \Omega \cup \underbrace{\Omega' \setminus \Omega}_{= \emptyset} \subseteq \Omega'$. $\square$



(a) The vertex v_4 of Ω lies inside J_e and will be replaced.

(b) Maximization in direction of other vertices.

(c) Replacement vertices u_1, u_2, u_3 and the sets J'_e and Ω' whose union is Ω.

(d) The resulting segment Ω'.

Figure 4.2: The computation steps of the cut-off operation. The vertex v_4 of the original segment Ω inside the jump enabling set J_e is replaced by the new vertices u_1, u_2, u_3. The vertices are obtained by maximizing from v_4 in the direction of the vertices v_1, v_2 and v_3 respectively.

To construct the over-approximation Ω' we therefore proceed as follows: First, we compute the vertices $\{v_1, \ldots, v_n\}$ of Ω. For each vertex v_i that is contained in J_e we replace v_i with the vertices $u_{i,j} = (1 - \lambda_{i,j})v_i + \lambda_{i,j}v_j$ for $j \neq i$ where $\lambda_{i,j}$ is maximal such that $u_{i,j} \in \Omega \cap J_e$. Note that pairs (i, j) can be skipped if both v_i and v_j are in J_e, because both vertices can be replaced if applying Lemma 4.4.1 iteratively. Additionally to computing the vertices of Ω, an upper bound on the number of linear optimization problems we need to solve is therefore given by n^2. This is however a generous approximation because in practice many vertices do not lie in J_e at all and pairs (i, j) with $v_i, v_j \in J_e$ are skipped as well. We illustrate the presented method in Figure 4.2.

Next we show that the method in fact computes the convex hull of $\Omega \setminus J_e$.

Lemma 4.4.3. *Let Ω' be obtained as described above. Then* $\Omega' = \mathrm{chull}(\mathrm{cl}(\Omega \setminus J_e))$.

Proof. Since $\Omega \setminus J_e \subseteq \Omega'$ and Ω' is convex and closed it follows by definition that $\mathrm{chull}(\mathrm{cl}(\Omega \setminus J_e)) \subseteq \Omega'$. On the other hand, every vertex of Ω' is by construction a limit point of $\Omega \setminus J_e$ and therefore contained in $\mathrm{cl}(\Omega \setminus J_e)$. As Ω' is the convex hull of its vertices it follow that $\mathrm{chull}(\mathrm{cl}(\Omega \setminus J_e)) \subseteq \Omega'$. □

In the following we refer to Ω' as the *cut-off* of Ω and J_e, motivated by the way that parts of J_e are cut off from Ω. This is to contrast it to the computation of the convex hull, even though both algorithms result in the same set. To integrate refinement levels into our analysis algorithm we extend the *Refined* function from the refinement tree to map a node to a subset of $\mathit{Urg} \times \{0, 1, 2\}$, where for (e, r) the number r is the refinement level of e. Similarly, analysis paths are extended to hold not only the refined transitions but also their level. Here, zero stands for the lowest refinement level where urgent transitions are treated as non-urgent, one represents the cut-off level which computes the cut-off as described above and two is the most precise level which means computing the set difference. To compute the time successors we compute the cut-off for each segment with each jump enabling set J_e if e is refined at level one and afterwards the set difference for each transition at level two. For refinement we first refine all suitable refinement candidates to the cut-off level and if that is not sufficient to verify safety, they are refined to the next level.

Note that this approach can also easily be extended to include arbitrarily many refinement levels: in general, we first refine every suitable candidate to a level r and then start refining to level $r + 1$ and so on.

Chapter 5

Experimental Results

The analysis algorithm for urgent LHA developed in this thesis has been implemented in the HyPro library [Sch19] and integrated in the corresponding tool HyDRA. In this chapter we want to evaluate their efficiency at the hand of several benchmark models. In particular, we want to compare the basic analysis algorithm described in Chapter 3 with the CEGAR algorithm from Chapter 4 and also the potential improvements described in Section 4.4. We start by briefly describing HyPro and some implementation details in Section 5.1, where we also introduce our benchmark suite. After that we analyze the performance of the improvements from Section 4.4 in Section 5.2 to obtain refinement strategies for the benchmark instances. We use these to compare the CEGAR algorithm with the non-CEGAR algorithm in Section 5.3.

5.1 Setup

In this section we briefly describe the HyPro library for which we implemented the presented analysis algorithm and introduce the benchmark suite which we use in Section 5.2 and Section 5.3 to test the algorithms and compare their efficiency.

The HyPro library. HyPro [Sch19] is a C++ library in which various state set representations and operations on them are implemented. These operations are used to implement a flowpipe-construction based reachability algorithm for linear hybrid automata in the HyDRA tool. Additional features include dedicated analysis algorithms for subclasses of LHA, parallelization support, decomposition of automata in *subspaces* and the partial path refinement algorithm which we sketched in Section 4.1. HyPro also allows to select between different linear optimization backends and between inexact and exact arithmetic.

For our implementation we used the box and $\mathcal{H}$-polytope representations provided by HyPro, which means that we mainly focused on the high level implementation of the analysis algorithms. In particular we did not implement the set difference operation for boxes and polytopes but used the implementation developed as part of [Amf21]. The CEGAR algorithm presented in this thesis has similarities to the partial path refinement algorithm [SÁ18] included in HyDRA, so the coarse structure of our implementation is similar as well. In particular we reused the implementation of a reachtree and extended it to implement refinement trees.

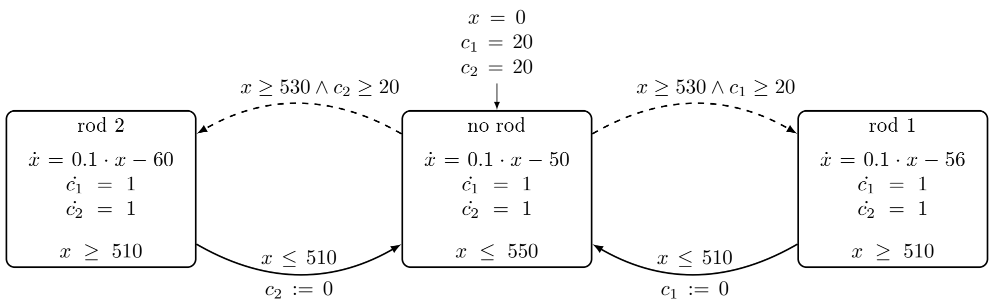


Figure 5.1: Rod reactor with urgency. Urgent transitions are indicated by dashed lines.

Benchmark suite. To analyze and test the developed algorithms we use example models of hybrid automata. While there are many models available to test reachability algorithms for LHA, there are few benchmarks that focus on urgency in hybrid systems. One of the few existing benchmarks of a batch reactor can be found in [MF14], however we were not able to verify this model with a non trivial jump depth in a reasonable time, as it is designed for algorithms focusing on LHA I.

We thus adapted existing models by adding and modifying urgent transitions. To ensure that refinement is necessary in most cases, in order to test the CEGAR algorithm, we experimentally adapted the sets of initial and bad states until they could not be verified without specialized treatment of urgent transitions. We now briefly describe each benchmark instance. The models can also be found in Appendix A.

Vehicle. This model is a variation of the vehicle automaton explained in Example 2.1.1 and Example 4.2.1. Here, we consider a vehicle represented by a point in x, y position moving with velocity v_x in x direction that starts at $x = 0$. At several regions constrained by the x, y position, the vehicle brakes for one second which is modeled by urgent transitions with the help of an addition clock variable. By choosing a large initial set for y we ensure that some splitting occurs when computing the set difference with these regions. By arranging the regions in a way such that every trace moves through at least one brake cycle we ensure that the vehicle cannot reach a position with $x = 15$ and maximal velocity which gives the safety specification.

Rod reactor. The rod reactor system [JLHM91, ACH$^+$95] models a simplified reactor core of a nuclear power plant and its temperature. One of two cooling rods can be inserted in the system which causes the temperature of the core to decrease. After a cooling rod has been inserted and removed from the system it cannot be inserted again for 20 seconds. The temperature x increases as long as no cooling rod is inserted and the safety specification is to verify that the temperature cannot exceed an upper limit of 550 while no cooling rods can be inserted.

To add urgency to the system we made the insertion of both rods urgent which means they are inserted whenever possible. The resulting automaton is depicted in Figure 5.1. In order to make the system unsafe if urgency is not respected, the unsafe state has been modified to $x > 550$, so the constraints $c_1 < 20$ and $c_2 < 20$ are dropped. Since HyPro does not currently support strict inequalities in bad state specifications this has been further approximated by $x \geq 550.1$.

Bouncing ball. The bouncing ball [JELS99] is a classical example for hybrid systems and models the height and vertical velocity of a ball that bounces on the ground. Each time the ball hits the ground it loses kinetic energy and slows down, whereas while falling it accelerates due to the gravitational force.

Here, we use a modified version of the bouncing ball with three instead of one dimension. The ball moves with a constant velocity in x direction and bounces in y direction where y is the height of the ball. The velocity in z direction is constantly zero. At $x = 0$ and $x = 2$ the x direction is instantly inverted and the ball again loses some of its kinetic energy in that direction. Intuitively, the ball therefore bounces in a room with walls at $x = 0$ and $x = 2$. Figure 5.2 shows plots for all benchmarks including the bouncing ball which illustrates this idea as well.

To introduce urgency in the system we assume that there is a horizontal beam positioned at $y = 1$ in the area $z \in [-0.5, 0.5]$. This is modeled by an urgent transition that forces the ball to bounce when the beam is hit while falling. This can cause fragmentation of the segments into the sets with $z \leq -0.5$ and $z \geq 0.5$, where the jump is not enabled and the set with $z \in [-0.5, 0.5]$. We therefore use an initial stateset with $z \in [-1, 1]$, so that the urgent transition in fact causes splitting. As safety specification we want to verify that the region below the beam at $x \in [1, 1.1]$ with $y = 0$ and $z = 0$ cannot be reached.

The bouncing ball with horizontal beam is adapted to a second instance, the bouncing ball with tilted beam. Here, the beam is tilted which seems like a minor difference at first but is actually relevant since the jump enabling set is not box shaped anymore as is the case with a horizontal beam.

In the following we call the first instance "BB horizontal" and the second "BB tilted".

Lawn mower. Finally, we consider two instances of a lawn mower benchmark which is adapted from the models presented in [ZSR$^+$12, pro10]. The original model is a probabilistic system in which a lawn mower, represented by x, y coordinates, moves on a lawn and randomly chooses between different x, y-velocities when it reaches the edge of the lawn. The locations of this model therefore represent the different directions in which the mower can move. Only when the edge of the lawn is reached, the mower switches directions which is modeled by discrete transitions to the other directions. As a safety specification it is assumed that a tarpaulin covers a region of the lawn which should not be reached.

To analyze the model we first replace all randomized transitions by assigning a probability of 1 to one of them. To add urgency we add an enclosing tolerance region around the tarpaulin, where the mower senses that it is close to the tarpaulin. If the region is entered, it immediately switches directions which is modeled by urgent transitions to the other locations, so that the region with the tarpaulin can never be reached. The tightness of the tolerance region can be used to adjust the difficulty of the benchmark.

We use two instances of the lawn mower benchmark. The first, which is referred to as "Lawn mower 1", has one unsafe set with an enclosing rectangular tolerance region. The second instance "Lawn mower 2" has an additional unsafe set with a polytopal, non-rectangular tolerance region. As initial states we use $[10, 10.5] \times [20, 20.5]$ for both instances.

We summarize the main characteristics of each benchmark instance in Table 5.1.

Table 5.1: Benchmark properties and parameters.

Instance	#*Var*	#*Loc*	#*Edge*	#*Urg*	Time horizon	Jump depth
Vehicle	4	2	7	6	20	∞
Rod reactor	3	3	4	2	20	5
BB horizontal	5	2	7	1	10	9
BB tilted	5	2	7	1	10	9
Lawn mower 1	2	4	16	8	50	6
Lawn mower 2	2	4	24	16	50	6

Settings. We analyze the benchmarks using the box and $\mathcal{H}$-polytope representations. For analysis with $\mathcal{H}$-polytopes it is necessary to convert between $\mathcal{H}$ and $\mathcal{V}$ polytopes and this operation tends to be numerically, so we use exact arithmetic. For boxes it would also be possible to use inexact arithmetic, i.e., floating point numbers which is generally faster. The comparison between results is however very similar as the relations don't change significantly and so we only include the running times with floating point numbers for boxes in the appendix in Table B.2.

We use aggregation for all benchmark instances and the time step sizes shown in Table 5.2. In some instances we use different time step sizes for boxes and polytopes so that safety can be verified with boxes while polytopes are still reasonably fast.

The results in the next sections were obtained on an Intel core i5-7200U at 2.50 GHz with 8 GB RAM. We set a time limit of 20 minutes and averaged the running times over five executions.

Table 5.2: Time step sizes used for verification. Aggregation is used for all instances.

Instance	Representation	Time step
Vehicle	Box	0.2
	HPol.	0.2
Rod reactor	Box	0.1
	HPol	0.1
BB horizontal	Box	0.001
	HPol	0.01
BB tilted	Box	0.001
	HPol	0.01
Lawn mower 1	Box	0.15
	HPol	0.15
Lawn mower 2	Box	0.1
	HPol	0.15

5.2 Refinement Strategies

We now discuss the results obtained from applying the developed algorithms to the benchmark suite. First we will compare different refinement strategies for the CEGAR algorithm by using the heuristics presented in Section 4.4. After that we will compare running times for using different refinement levels which also means that we compare the CEGAR approach with the non-CEGAR approach where urgency is always considered.

Table 5.3: Verification times in seconds with different heuristics with and without pruning of segments contained in urgent jump enabling sets. The numbers in brackets are the number of refinement iterations. Heuristics are None (default strategy), Count, Vol. (Volume) and CC (Constraint-count). Timeout (TO) is 20 minutes.

			Heuristic			
Instance	Rep.	Prune	None	Count	Vol.	CC
Vehicle	Box	No	3.17 (4)	3.10 (4)	2.35 (2)	3.15 (4)
		Yes	3.02 (3)	3.07 (3)	2.22 (1)	3.00 (3)
	HPol	No	290.75 (6)	289.05 (6)	215.56 (2)	288.96 (6)
		Yes	264.60 (6)	264.24 (6)	181.85 (1)	266.54 (6)
Rod reactor	Box	No	1.23 (7)	1.43 (8)	1.65 (8)	1.23 (7)
		Yes	1.39 (7)	1.62 (8)	1.83 (8)	1.39 (7)
	HPol	No	17.48 (7)	20.15 (8)	20.99 (8)	17.47 (7)
		Yes	20.03 (7)	22.60 (8)	23.77 (8)	19.92 (7)
BB horizontal	Box	No	4.28 (1)	4.30 (1)	4.36 (1)	4.32 (1)
		Yes	4.43 (1)	4.54 (1)	4.52 (1)	4.44 (1)
	HPol	No	286.97 (1)	286.65 (1)	287.72 (1)	289.09 (1)
		Yes	288.35 (1)	291.01 (1)	292.05 (1)	288.85 (1)
BB tilted	Box	No	6.39 (2)	6.43 (2)	6.36 (2)	6.34 (2)
		Yes	6.77 (2)	6.75 (2)	6.77 (2)	6.74 (2)
	HPol	No	349.25 (1)	349.82 (1)	349.23 (1)	349.77 (1)
		Yes	350.67 (1)	350.53 (1)	352.27 (1)	352.34 (1)
Lawn mower 1	Box	No	10.39 (506)	3.92 (228)	43.74 (1646)	10.46 (506)
		Yes	2.89 (2)	2.85 (2)	2.84 (2)	2.85 (2)
	HPol	No	242.77 (468)	66.84 (90)	190.53 (186)	243.68 (468)
		Yes	73.35 (0)	74.09 (0)	73.55 (0)	73.72 (0)
Lawn mower 2	Box	No	191.73 (2013)	113.18 (1130)	TO (4900)	189.75 (3205)
		Yes	189.90 (1702)	181.12 (1418)	TO (4495)	191.28 (2766)
	HPol	No	TO (1758)	408.31 (495)	TO (1284)	TO (1819)
		Yes	447.78 (86)	388.50 (47)	569.73 (124)	407.74 (66)

Pruning segments. The running times for the selected benchmarks when choosing different heuristics are listed in Table 5.3. We additionally compare the running times between enabled and disabled pruning of segments contained in the jump enabling set of an urgent transition. We first discuss the impact of pruning and then each heuristic in more detail.

While pruning improves the precision it introduces computational overhead, because we need to check for every segment Ω and jump enabling set J_e whether $\Omega \subseteq J_e$, which is done by computing the intersection $\Omega \cap J_e$ and comparing it to Ω. In practice the intersection can be done in two steps: First, we intersect the segment with the guard $\Omega \cap g_e$ and only if the result is equal to Ω the intersection with J_e is computed. The advantage is that the result of the first intersection can be reused as jump predecessor for the computation of discrete successors. However, for the nodes of maximal depth, i.e., when the maximal jump depth is reached, the intersection is not reused which may result in unnecessary intersection steps.

We observe this especially in the rod reactor and bouncing ball benchmarks where only few segments at the end of flowpipes could be pruned. In these cases the computational overhead outweighs the benefits.

For the vehicle benchmark, pruning is beneficial for all refinement strategies because more segments are contained in jump enabling sets. The improvement here is more significant for polytopes, with a speedup of about 1.1, where time successor computation is slower and more segments can be pruned due to higher fragmentation. For boxes, the highest observed speedup is 1.05 so the running times are almost identical.

Pruning is very advantageous for the lawn mower benchmark. In particular for the first instance no further refinement is necessary when using polytopes as state set representations and only two additional refinement iterations are necessary for boxes, both of which use the same refinement candidate. This explains the considerable performance improvement with a speedup of 3.6 for boxes and 3.3 for polytopes without using refinement heuristics. Only when using the count heuristic with polytopes, pruning segments is slightly worse for this benchmark, which can only be explained by the discussed computational overhead for nodes of maximal depth. This overhead can be significant here because the lawn mower benchmark has a relatively high branching factor in the reachtree and thus many nodes at maximal depth.

Similarly, in the second lawn mower instance pruning segments is a considerable improvement for polytopes, so much that without pruning only one heuristic is able to verify safety within the time limit, whereas with pruning all strategies can verify safety. For boxes, pruning is again less impactful and worse when using the count heuristic. Interestingly, pruning caused more refinement iterations in this last case, with 1418 iterations with pruning and 1130 without. This suggests that pruning can result in a worse refinement strategy. Specifically we observed that more "nested" refinements were necessary by which we mean cases where refuting a single counterexample required multiple refinement iterations. It is somewhat difficult to trace why this happens, but a possible explanation is that it is sometimes better to refine early, which increases the precision in all descending subtrees, than continuing computation and refining multiple paths later. Without pruning we may encounter an unsafe state earlier, which means that more refinement needs to be done early but consequently fewer refinement iterations are necessary later compared to enabled pruning. Since the accumulated error along paths is more significant for boxes than for polytopes, mostly due to higher imprecision in the union operation, this could also explain why pruning is less advantageous for boxes than for polytopes.

All in all, pruning segments is an improvement in most cases, especially for polytopes where it only causes slight computational overhead in some instances. On the other hand we have also seen that pruning can result in a worse refinement strategy where more refinement iterations are necessary in the long run, especially when using over-approximating representations such as boxes.

Heuristics. We now discuss the effect of the different refinement heuristics in more detail. Here we ignore the bouncing ball benchmark since in all instances at most two refinement iterations are done which means that heuristics have essentially no effect. Additionally, it should be noted that in all benchmark instances except the second lawn mower benchmark all guards of urgent transitions have the same number of constraints, which means that the constraint count heuristic behaves the same as using the default ordering without heuristics.

For the vehicle benchmark the volume heuristic performs best with a speedup of around 1.3 for both representations compared to using no heuristic. This is because to verify safety it is sufficient for every trace to move through one brake cycle. By picking the largest regions first, fewer regions need to be considered to cover every trace, which results in fewer refinement iterations. The count heuristic has no impact on this instance since no transition is refined more than once.

Both heuristics lead to a slightly worse refinement strategy when used on the rod reactor benchmark, as they cause more refinement iterations. While the volume heuristic finds suitable candidates in fewer iterations it has the additional overhead of computing the intersection-volumes which takes around 10% of the running time in this instance.

In the lawn mower benchmarks we see the biggest impact of the heuristics since the number of refinement iterations is much greater in both instances. We can see that the count heuristic is a major improvement with a speedup of 2.6 for boxes and 3.6 for polytopes in the first instance without segment pruning. In the second instance the improvement is less significant for boxes with a speedup of 1.6 without pruning and almost the same running times with pruning. For polytopes without pruning the count heuristic is the only strategy that can verify safety within the time limit and gives a slight performance boost with pruning enabled. The primary reason why the count heuristic performs so well on this benchmark is that many of the urgent transitions have the same guards. The count heuristic essentially only tests one "representative" transition with the highest counter for each of the guards and ignores the other transitions.

Interestingly, the volume heuristic performs very poorly on the lawn mower benchmarks when using boxes as state set representations and is somewhat volatile for polytopes. While the overhead of computing the volume is a contributing factor as it takes up to 50% of the running time for boxes, the heuristic also causes a worse refinement strategy here - even if volume computation was instant, using it would still be considerably slower compared to the default strategy. An explanation for this is that the volume heuristic tends to refine nodes closer to the counterexample node first, because the segments are larger due to accumulated error and thus nodes with higher depth are considered first as candidates. This is problematic when multiple refinements are necessary to refute a counterexample since more backtracking has to be done as already illustrated in Example 4.3.3. Indeed, looking at the first instance, it took 9.9 refinement iterations on average to refute a single counterexample when using the volume heuristic and the average distance between the unsafe node and the chosen refinement candidate was 1.55. In contrast, it only took 5.7 iterations on average without using heuristics, where the average distance was considerably higher at 2.12. This means that more backtracking was done when using the volume heuristic. This is not the case for polytopes where the distance between refinement node and unsafe node is still lower with an average of 1.5 versus 1.9, but the average number of refinement iterations is lower with 2.5 versus 6.2 without heuristics. The difference between boxes and polytopes can be explained by the large over-approximation caused by using boxes: to see why, note that in some cases it is actually beneficial to refine nodes close to the counterexample node first, since the path to the unsafe node is shorter. This is true if only few refinement steps are necessary to refute a counterexample and thus no backtracking is done, which is more likely to occur with polytopes than with boxes as they are more precise.

Finally, we consider the constraint count heuristic for the second lawn mower instance. Here, it makes a difference as one of the jump enabling regions has 4 constraints while the other has 8. Thus the first one is prioritized with the goal of causing fewer splits in the set difference operations. This goal is achieved for all considered instances, for example using boxes without pruning we have on average

2.56 fragments if set difference causes a split and 2.66 without using the constraint count heuristic. For polytopes with pruning the difference is more significant with an average of 2.12 fragments with heuristic and 2.46 without. For boxes this is however not a huge performance boost and is mostly offset by more total refinement iterations. This is similar to the case of the volume heuristic where nodes at higher depth are prioritized for refinement since this is when the transitions with fewer constraints are encountered more often.

Conclusion. We can see that the optimal refinement strategy depends heavily on the system and even on the chosen representation. The volume heuristic tends to find suitable refinement candidates in few iterations and can improve the strategy when only few refinement iterations are necessary. On the flip side it leads to significant backtracking when that is not the case and also has a considerable computational overhead which often offsets its advantages.

In contrast, the count heuristic shines when many refinement iterations are necessary as previous results can often be reused. The count heuristic should also be considered when multiple transitions have similar or even the same jump enabling sets, since it naturally leads to "representative" transitions for each of these sets rather than testing each individually.

More testing should be done for the constraint count heuristic, but from our experiments it seems to reduce the splitting caused by set difference computation. However, it remains to be seen whether this is relevant in practice, since many systems tend to have guards with one or only few constraints where the heuristic makes little difference and from our experiments it seems that a good refinement strategy may be more impactful.

5.3 Refinement Levels

We now compare the different refinement levels with each other. Recall that we have three refinement levels for each transition in each node: The first and least precise is called the "unrefined" level, which does not consider urgency at all. The second level, described in detail in Section 4.4, we call the "cut-off" level and it partially handles urgency by taking the convex hull of the set difference of a segment with the jump enabling sets of urgent transitions. Finally, we have the third "set difference" level which computes the set difference and can cause splitting of segments into multiple fragments.

By enabling only the set difference level we get a non-CEGAR algorithm that computes the reachable sets of urgent LHA. Plots created by using this algorithm on our benchmark suite are shown in Figure 5.2 with boxes as representation. Plots with polytopes as representations are shown in Figure 5.3 for benchmark instances where the increased precision of polytopes is significant. For the CEGAR algorithm we include the plots in Figure 5.4, where we omit flowpipes of spurious counterexample nodes which means in particular that no unsafe segments are shown.

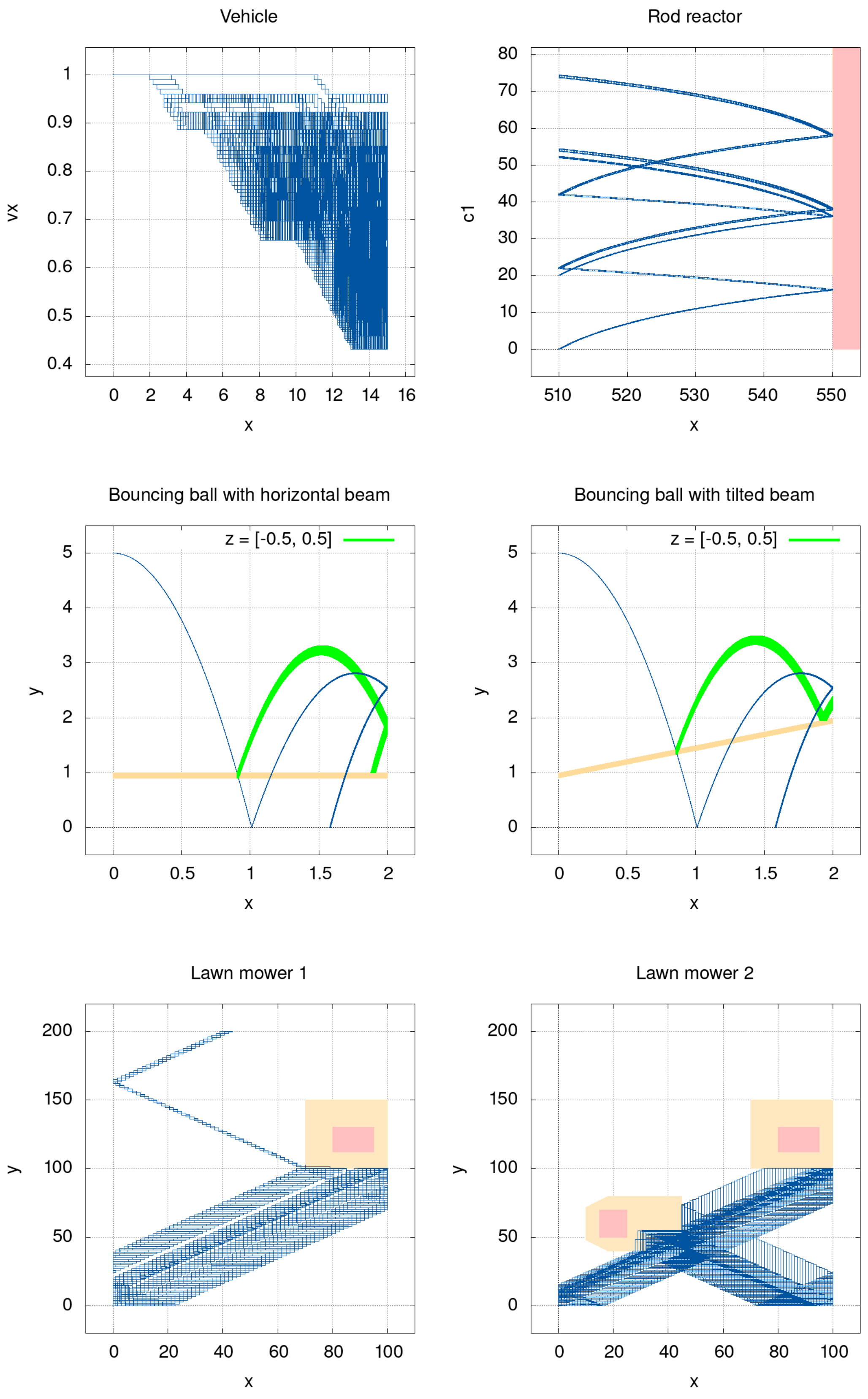


Figure 5.2: Plots for the benchmark instances with box representation. The jump depth for the bouncing ball benchmarks is reduced to 3 and for the second lawn mower instance to 4. Red regions indicate unsafety and orange regions are urgent jump enabling sets. In the plots for the bouncing ball, the orange beam and green segments are exclusive to $z \in [-0.5, 0.5]$ while segments below the beam are exclusive to $z \notin [-0.5, 0.5]$.

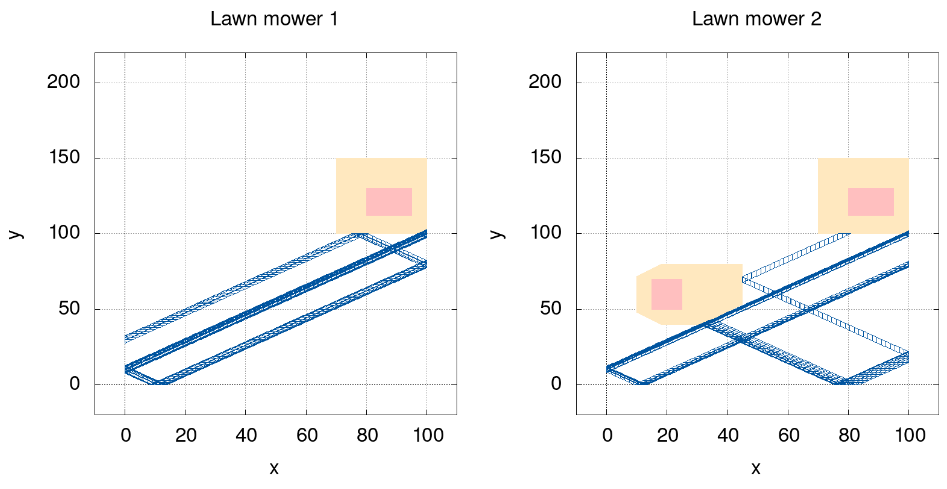


Figure 5.3: Plots for the lawn mower benchmarks with polytope representation. The jump depth of the second instance is reduced to 4.

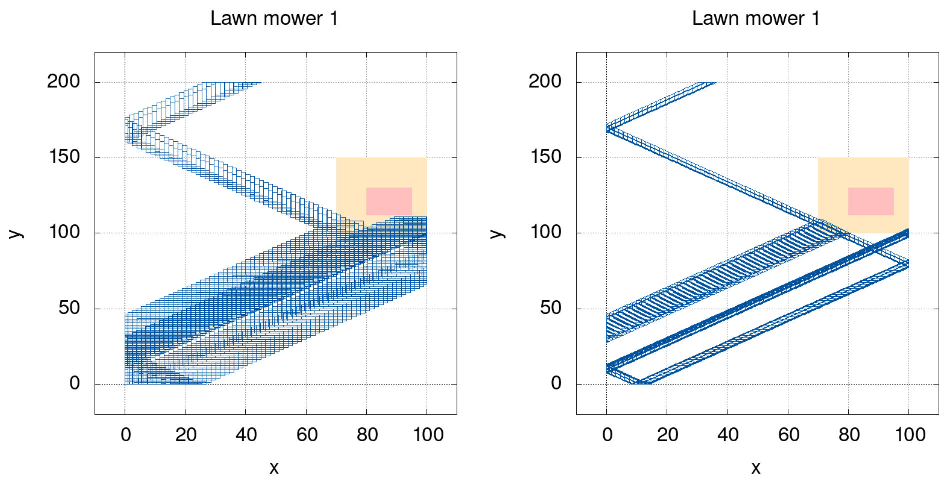


Figure 5.4: Plots generated by the CEGAR algorithm for the first lawn mower benchmark with box representation (left) and polytope representation (right). Flowpipes of refuted counterexamples are not shown.

Table 5.5: Verification times in seconds with different enabled refinement levels. Refinement levels are labeled SD (set difference), C (cut-off) and U (unrefined). The used heuristic for the CEGAR algorithms are listed in Table 5.4. Timeout (TO) is 20 minutes.

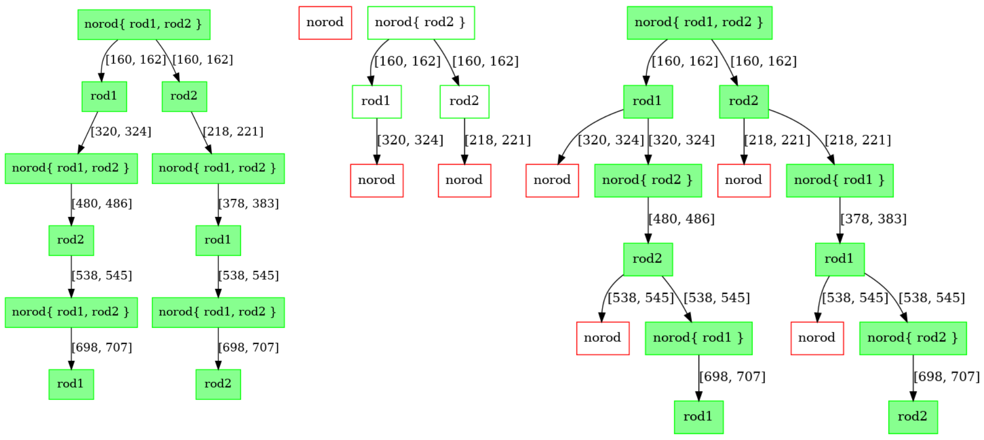
			Refinement levels		
Instance	Rep.	Prune	SD	U, SD	U, C, SD
Vehicle	Box	No	4.24	2.35	2.59
		Yes		2.22	2.33
	HPol	No	TO	215.56	208.80
		Yes		181.85	180.07
Rod reactor	Box	No	0.68	1.23	1.85
		Yes		1.39	2.00
	HPol	No	10.38	17.48	22.42
		Yes		20.03	24.98
BB horizontal	Box	No	3.44	4.28	6.71
		Yes		4.43	6.80
	HPol	No	TO	286.97	295.40
		Yes		288.35	292.83
BB tilted	Box	No	TO	6.39	11.66
		Yes		6.77	11.93
	HPol	No	TO	349.25	350.02
		Yes		350.67	351.33
Lawn mower 1	Box	No	4.28	3.92	4.13
		Yes		2.85	2.85
	HPol	No	132.97	66.84	47.71
		Yes		74.09	74.02
Lawn mower 2	Box	No	272.24	113.18	297.90
		Yes		181.12	301.57
	HPol	No	544.13	408.31	461.01
		Yes		388.50	372.91

Table 5.4: Selected heuristics for each benchmark.

Instance	Heuristic
Vehicle	Volume
Rod reactor	None
BB horizontal	None
BB tilted	None
Lawn mower 1	Count
Lawn mower 2	Count

In Table 5.5 the running times of the non-CEGAR algorithm are compared with the running times of the CEGAR algorithm and with the addition of the cut-off level. In order to keep the table at a reasonable size we use a selected refinement heuristic for each benchmark. The selection is based on the results from Section 5.2 and is listed in Table 5.4. Exhaustive tables with all possible combinations can be found in Appendix B.

We first compare the non-CEGAR algorithm with the CEGAR algorithm, i.e., the refinement levels SD and U, SD in Table 5.5. In the vehicle benchmark we note that it is feasible to not use CEGAR with boxes while for polytopes safety could only be verified in time



(a) Non-CEGAR algorithm refinement tree.

(b) CEGAR algorithm refinement trees.

Figure 5.5: Refinement trees generated for the rod reactor benchmark by the non-CEGAR and CEGAR algorithms.

when using CEGAR. This means that CEGAR is a much more considerable improvement for polytopes than for boxes in this instance. The reason why the non-CEGAR algorithm is reasonably fast for boxes but slow for polytopes is that all jump enabling sets are box shaped in this instance and we observed that using the box-based set difference algorithm causes considerably less splitting than the polytopal set difference algorithm. This is likely also caused by more optimization in the box algorithm as it is in general less complex. In fact, only 0.08% of set difference operations resulted in more than one fragment for boxes while it was almost twenty times as high for polytopes with 1.5% of the set difference operations resulting in multiple fragments. Using CEGAR the splitting rate for polytopes is almost as high with 1.2% but fewer total set difference operations are necessary than without CEGAR: while without CEGAR more than 120 000 set difference operations are executed before the timeout only 1499 are needed in the CEGAR algorithm, which highlights the exponential blowup that set difference can cause.

In the rod reactor benchmark CEGAR is slower for both representations for which there are two reasons. The first is that even in the non-CEGAR algorithm, set difference never results in multiple fragments, which negates our motivation for using CEGAR in the first place. Additionally, all set difference computations are necessary to verify safety which means that the CEGAR algorithm involves the computation of the fully refined refinement tree. The refinement trees created by both algorithms are shown in Figure 5.5. Here, we denote the refined transitions in each node by their target location, which means that for example in the root node in (a) the two outgoing urgent transitions to *rod1* and *rod2* are refined. Note that (b) depicts multiple refinement trees since the root node is refined multiple times. Here, the subtree generated by the highlighted nodes induces the same flowpipes as the non-CEGAR refinement tree in (a), where the additional refined transitions at depths 2 and 4 have no effect as the intersection of their jump enabling sets is empty with all segments. Thus the CEGAR algorithm essentially has the non-CEGAR algorithm as a subroutine which makes it obvious that it should be slower.

On the bouncing ball benchmark a similar effect to the vehicle benchmark can be observed, where verification is even slower for boxes with CEGAR than without, while CEGAR is again a major improvement for polytopes where without CEGAR safety cannot be verified within the time limit. The reason is again that the jump enabling set of the urgent transition is box shaped, so set difference causes a lot less splitting for boxes than for polytopes. However, switching to a non box shaped jump enabling set in the instance with the tilted beam we observe that even with boxes the time limit was exceeded without CEGAR, while CEGAR barely slows down. In this instance it was necessary to use the polytope based set difference algorithm for boxes to assure over-approximation (see "Boxes" on Page 19), which causes exponential splitting.

The same effect can be observed between the two lawn mower benchmarks: While the first instance with boxes can be verified almost as fast without CEGAR since it only has rectangular jump enabling sets, CEGAR causes a significant speedup of 2.4 in the second instance, where an additional non-rectangular jump enabling set is present. Because the lawn mower benchmarks are only two-dimensional and even polytopal set difference doesn't cause too much splitting here, both instances can also be verified without CEGAR and with polytopes, although CEGAR is still faster.

Finally, we want to evaluate the addition of the cut-off refinement level, where the motivation was to obtain a convex over-approximation of the set difference to negate the exponential blowup in the number of segments. Note that the cut-off operation doesn't have any effect in the bouncing ball benchmarks, so the same refinement steps as without cut-off level still need to be done. It is therefore unsurprising that using the cut-off level in this benchmark is always slower.

Unfortunately, it also has an adverse effect on all instances with boxes, even though the cut-off level is enough to verify the rod reactor, vehicle and first lawn mower benchmarks, i.e., no set difference was computed in these instances. The main reason is that the cut-off operation itself is slower than set difference computation and set difference doesn't create a lot of fragments for boxes. For example a single cut-off computation took 1.32 ms on average in the vehicle benchmark while a set difference operation took 0.07 ms on average. Since few fragments are created and additionally time elapse is extremely fast for boxes, this computational overhead is not worth it. The worst effect can be observed in the second lawn mower benchmark. Here the cut-off operation is not sufficient to verify safety, so that multiple transitions need to be refined to the last level which causes a lot of backtracking.

For polytopes on the other hand we have seen that splitting is more of an issue, so the additional refinement level may be an improvement in some cases. But, same as for boxes the cut-off operation is generally slower than computing the set difference. Thus if set difference doesn't cause any splits such as in the rod reactor benchmark it is expected that adding the cut-off level is disadvantageuous. Here, a cut-off operation took 3.59 ms on average while a set difference operation only took 0.66 ms on average. On the other benchmarks, cut-off has a mostly positive effect with polytopes, especially in the vehicle and first lawn mower benchmark. Here, the most significant speedup of 1.39 is obtained in the first lawn mower instance without pruning. On the second lawn mower benchmark cut-off is again slower because while it is enough to verify safety, more refinement iterations are necessary than when taking the set difference (664 vs. 495 iterations).

Conclusion. We have seen that CEGAR can be a vast improvement when the computation of set difference causes a lot of splitting. This mostly happens when polytopal set difference computation is necessary, e.g., when the urgent jump enabling sets are not box shaped or when the $\mathcal{H}$-polytope representation is used. However, this is not a necessary condition as we have seen on the rod reactor benchmark, where no splitting is caused. This shows that it is rarely easy to say beforehand on which models CEGAR will perform better. Note also that we used the results from Section 5.2 to determine the optimal heuristic beforehand and that choosing a bad heuristic can make a huge difference as well.

The addition of the cut-off level was only an improvement for polytopes in some instances. This mainly stems from the fact that computing the cut-off is generally much slower than computing the set difference which means that it is only advantageous if set difference would cause a lot of splitting. Additionally, we need to consider that more refinement iterations may be necessary since the cut-off operation is less precise than the set difference operation. Even more backtracking can be caused if the cut-off level is not sufficient to verify safety. However, when this is not the case and polytopes are used as state set representation then the cut-off level can be beneficial since it does reduce the number of segments that are computed.

Chapter 6

Conclusion

6.1 Summary

In this thesis we extended hybrid automata with a set of urgent transitions in order to accurately model a larger set of hybrid systems. For linear hybrid automata we have developed an adaption of the flowpipe-construction algorithm and have also seen a specialized flowpipe-construction algorithm for urgent LHA I, which is able to compute the set of reachable states exactly. For both algorithms we used the set difference operation to construct the flowpipe segments, which can split the segments into multiple convex fragments. To mitigate the exponential increase in the number of segments this can cause we applied the CEGAR technique by respecting the urgency of transitions only on demand.

We introduced refinement trees and analysis paths as a formalization for the CEGAR algorithm in order to dynamically refine individual nodes without restarting analysis after refinement while also reusing previous computation results. This resulted in an analysis algorithm for urgent LHA that can be adjusted by choosing a refinement strategy that constructs the analysis path to be refined. We realized multiple strategies by introducing different heuristics for choosing refinement candidates as a combination of node and urgent transition on unsafe paths. Finally, we presented an alternative method for computing the convex hull of the set difference, which is more suited for $\mathcal{H}$-polytopes. This "cut-off" operation was used to define a third middle ground refinement level for transitions which we integrated into the refinement algorithm.

Our experimental results showed that the refinement strategy can have a significant impact on the performance of the CEGAR algorithm by causing more or fewer refinement iterations. An important step of applying the CEGAR algorithm is therefore to choose a suitable refinement strategy: for our selected heuristics we could observe that the "count" heuristic performs well when urgent jumps have similar guards or when many refinement iterations are necessary. The "volume" heuristic on the other hand is better suited when few refinement iterations are sufficient to verify safety, although it causes overhead due to the volume computation. The additional cut-off refinement level was not an improvement in most cases, which is mainly because the cut-off operation is still much slower than the set difference operation, and this can not always be outweighed by the reduced splitting.

Compared to an analysis algorithm without refinement we have seen that the developed CEGAR algorithm can save a lot of time provided that set difference causes significant splitting. However, if this is not the case or when many transitions need to be refined and thus lots of backtracking is necessary, CEGAR tends to perform worse. A challenge is therefore that it is not always clear beforehand whether urgency causes redundant splitting or not, and consequently whether the CEGAR approach is beneficial. In the current implementation we could observe that this is more often the case when polytopal set difference has to be used, which may be necessary when jump enabling sets are not box shaped or when a more precise state set representation than boxes is required.

6.2 Future Work

Regarding the analysis of urgent hybrid automata, there is some room for improvement by making an additional effort to exclude the shadows of jump enabling sets. In this thesis we briefly sketched an approach based on connected components in a graph where the nodes represent segments, however this is not yet implemented in HyPro. Excluding the shadow could make analysis more precise and faster by excluding fragments caused by the set difference operation. Similarly, the specialized algorithm for urgent LHA I presented in this thesis can be implemented in HyPro and is expected to perform much better on LHA I. Additionally, there may be room for improvement in this algorithm considering the handling of multiple urgent transitions. Here, we only suggested handling this by computing the pairwise set difference of their shadows, but simpler and more efficient methods that consider the other urgent transitions already when computing the jump predecessors may be possible. Once the approach is implemented, it can easily be integrated into the same CEGAR algorithm we presented in this thesis, although it is unclear whether that would be beneficial as possibly fewer splits are caused by the set difference during flowpipe constructing for LHA I.

It could also be beneficial to attempt further optimization of the set difference algorithms, especially for polytopes as this currently causes a lot of splitting. One direction that future work could take in this area is the ordering of halfspaces in the set difference algorithm, as we have seen that this can have an impact on the number of created fragments. Generally, it may also be helpful for optimizations to utilize strict inequalities which are currently not widely supported in HyPro.

Finally, the CEGAR algorithm can be extended in multiple ways. The first is adding more refinement strategies, which can be done easily by adjusting the heuristics, but more elaborate strategies can be integrated as well. Additionally, the algorithm can possibly be improved by adding different refinement levels, similar to the presented cut-off level. In particular, while the cut-off level reduces splitting, it has the disadvantage that the cut-off operation is very slow. Future refinement levels could therefore aim at giving a coarser convex over-approximation of the set difference that can quickly be computed, since we currently use the most precise one, i.e., the convex hull. A possible idea for such an operation is to use *templates* [Sch19], which can be used to over-approximate sets with a fixed number of halfspaces. While less precise, the advantage of such an operation would be that computation of the vertices is not necessary, so that it would be faster especially for $\mathcal{H}$-polytopes.

Bibliography

[Ábr17] Erika Ábrahám. Modeling and analysis of hybrid systems. RWTH Aachen University, Lecture Notes, 2017.

[ACH+95] Rajeev Alur, Costas Courcoubetis, Nicolas Halbwachs, Thomas A. Henzinger, Pei-Hsin Ho, Xavier Nicollin, Alfredo Olivero, Joseph Sifakis, and Sergio Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[ADI03] Rajeev Alur, Thao Dang, and Franjo Ivancic. Counter-example guided predicate abstraction of hybrid systems. In *TACAS'03*, volume 2619 of *LNCS*, pages 208–223. Springer, 2003.

[Amf21] Kim Amfaldern. *Computing set difference for the reachability analysis of hybrid systems.* Bachelor's thesis, RWTH Aachen University, 2021. Unpublished thesis.

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

[Bao09] Mato Baotic. Polytopic computations in constrained optimal control. *Automatika*, 50(3-4):119–134, 2009.

[BFT04] Alberto Bemporad, Carlo Filippi, and Fabio Danilo Torrisi. Inner and outer approximations of polytopes using boxes. *Computational Geometry*, 27(2):151–178, 2004.

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

[BMMW15] Sergiy Bogomolov, Daniele Magazzeni, Stefano Minopoli, and Martin Wehrle. PDDL+ planning with hybrid automata: Foundations of translating must behavior. In *Proc. of ICAPS'15*, pages 42–46. AAAI Press, 2015.

[BMPW14] Sergiy Bogomolov, Daniele Magazzeni, Andreas Podelski, and Martin Wehrle. Planning as model checking in hybrid domains. In *Proc. of AI'14*, volume 28, pages 2228–2234. AAAI Press, 2014.

[Buc43] Robert Creighton Buck. Partition of space. *The American Mathematical Monthly*, 50(9):541–544, 1943.

[CFH$^+$03] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14(4):583–604, 2003.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *CAV'00*, volume 1855 of *LNCS*, pages 154–169. Springer, 2000.

[CK98] Alongkrit Chutinan and Bruce H Krogh. Computing polyhedral approximations to flow pipes for dynamic systems. In *Proc. of CDC'98*, volume 2, pages 2089–2094. IEEE Computer Society, 1998.

[DKL07] Henning Dierks, Sebastian Kupferschmid, and Kim Guldstrand Larsen. Automatic abstraction refinement for timed automata. In *FORMATS'07*, volume 4763 of *LNCS*, pages 114–129. Springer, 2007.

[Far02] Julius Farkas. Theorie der einfachen ungleichungen. *Crelle*, 1902(124):1–27, 1902.

[FCJK05] Ansgar Fehnker, Edmund M. Clarke, Sumit Kumar Jha, and Bruce H. Krogh. Refining abstractions of hybrid systems using counterexample fragments. In *HSCC'05*, volume 3414 of *LNCS*, pages 242–257. Springer, 2005.

[FL06] Maria Fox and Derek Long. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *HSCC'05*, volume 3414 of *LNCS*, pages 291–305. Springer, 2005.

[Grü03] Grünbaum. *Convex polytopes*, volume 221 of *Graduate Texts in Mathematics*. Springer, 2003.

[Gue09] Colas Le Guernic. *Reachability Analysis of Hybrid Systems with Linear Continuous Dynamics*. PhD thesis, Joseph Fourier University, Grenoble, France, 2009.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proc. of LICS'96*, pages 278–292. IEEE Computer Society, 1996.

[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.

[JCKK18] Luc Jaulin, Martine Ceberio, Olga Kosheleva, and Vladik Kreinovich. How to efficiently compute ranges over a difference between boxes, with applications to underwater localization. Technical report, The University of Texas at El Paso, 2018.

[JELS99] Karl Henrik Johansson, Magnus Egerstedt, John Lygeros, and Shankar Sastry. On the regularization of zeno hybrid automata. *Systems & control letters*, 38(3):141–150, 1999.

[JKWC07] Sumit Kumar Jha, Bruce H. Krogh, James E. Weimer, and Edmund M. Clarke. Reachability for linear hybrid automata using iterative relaxation abstraction. In *HSCC'07*, volume 4416 of *LNCS*, pages 287–300. Springer, 2007.

[JLHM91] Matthew S. Jaffe, Nancy G. Leveson, Mats Per Erik Heimdahl, and Bonnie E. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, 1991.

[MF14] Stefano Minopoli and Goran Frehse. Non-convex invariants and urgency conditions on linear hybrid automata. In *FORMATS'14*, volume 8711 of *LNCS*, pages 176–190. Springer, 2014.

[MF16a] Stefano Minopoli and Goran Frehse. From simulation models to hybrid automata using urgency and relaxation. In *Proc. of HSCC'16*, pages 287–296. ACM, 2016.

[MF16b] Stefano Minopoli and Goran Frehse. SL2SX translator: From simulink to spaceex models. In *Proc. of HSCC'16*, pages 93–98. ACM, 2016.

[NÁ12] Johanna Nellen and Erika Ábrahám. Hybrid sequential function charts. In *MBMV'12*, volume 68 of *Forschungsergebnisse zur Informatik*, pages 109–120. Verlag Dr. Kovač, 2012.

[NÁW15] Johanna Nellen, Erika Ábrahám, and Benedikt Wolters. A CEGAR tool for the reachability analysis of PLC-controlled plants using hybrid automata. In *Formalisms for Reuse and Systems Integration*, volume 346 of *Advances in Intelligent Systems and Computing*, pages 55–78. Springer, 2015.

[pro10] ProHVer case studies. `https://depend.cs.uni-saarland.de/tools/prohver/casestudies/`, 2010. Accessed: 2021-09-19.

[RKML06] Sasa V. Rakovic, Eric C. Kerrigan, David Q. Mayne, and John Lygeros. Reachability analysis of discrete-time systems with disturbances. *IEEE Transactions on Automatic Control*, 51(4):546–561, 2006.

[SÁ18] Stefan Schupp and Erika Ábrahám. Efficient dynamic error reduction for hybrid systems reachability analysis. In *TACAS'18*, volume 10806 of *LNCS*, pages 287–302. Springer, 2018.

[Sch19] Stefan Schupp. *State set representations and their usage in the reachability analysis of hybrid systems.* Dissertation, RWTH Aachen University, 2019.

[SJ12] Peter Schrammel and Bertrand Jeannet. From hybrid data-flow languages to hybrid automata: a complete translation. In *Proc. of HSCC'12*, pages 167–176. ACM, 2012.

[Zie95] Günter M Ziegler. *Lectures on polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer, 1995.

[ZSR$^+$12] Lijun Zhang, Zhikun She, Stefan Ratschan, Holger Hermanns, and Ernst Moritz Hahn. Safety verification for probabilistic hybrid systems. *European Journal of Control*, 18(6):572–587, 2012.

Appendix A

Benchmark Automata

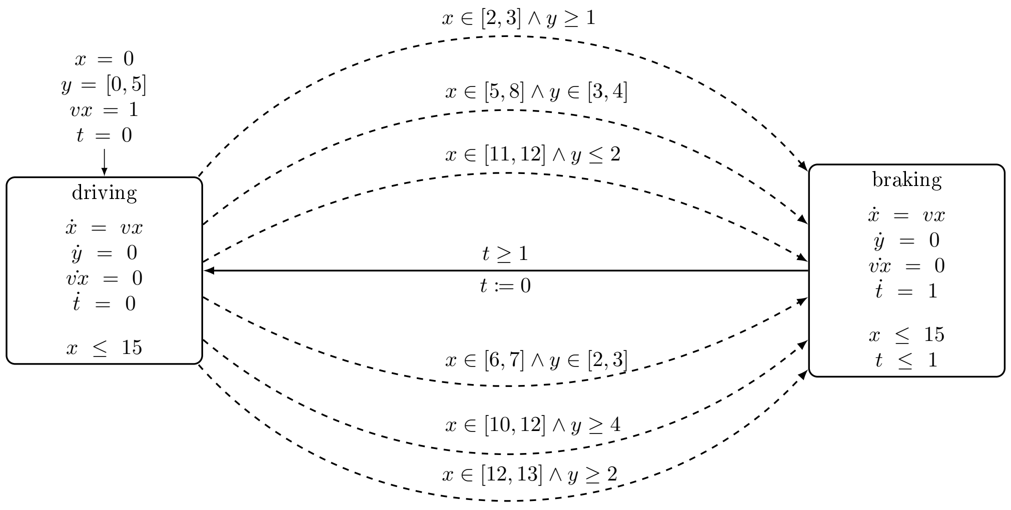


Figure A.1: Vehicle. Urgent edges are indicated by dashed lines.

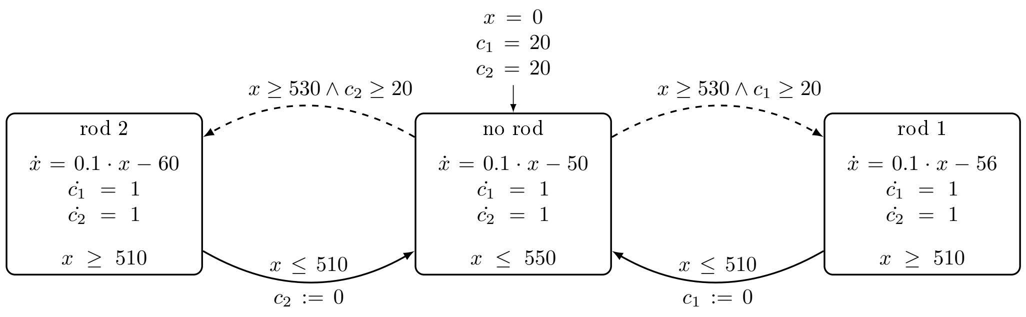


Figure A.2: Rod reactor. Urgent edges are indicated by dashed lines.

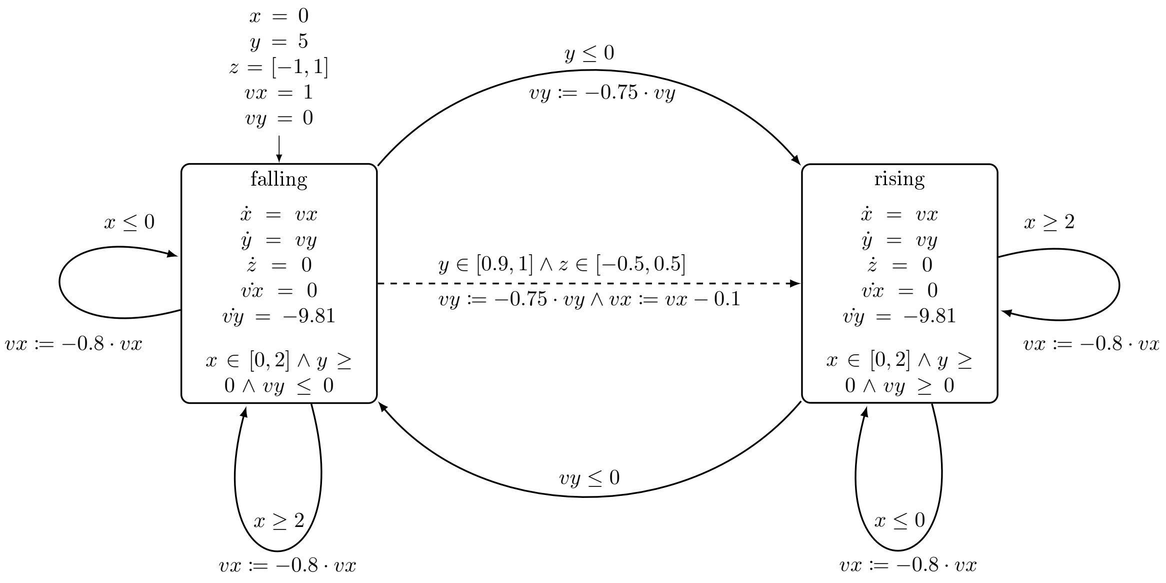


Figure A.3: Bouncing ball with horizontal beam. Urgent edges are indicated by dashed lines.

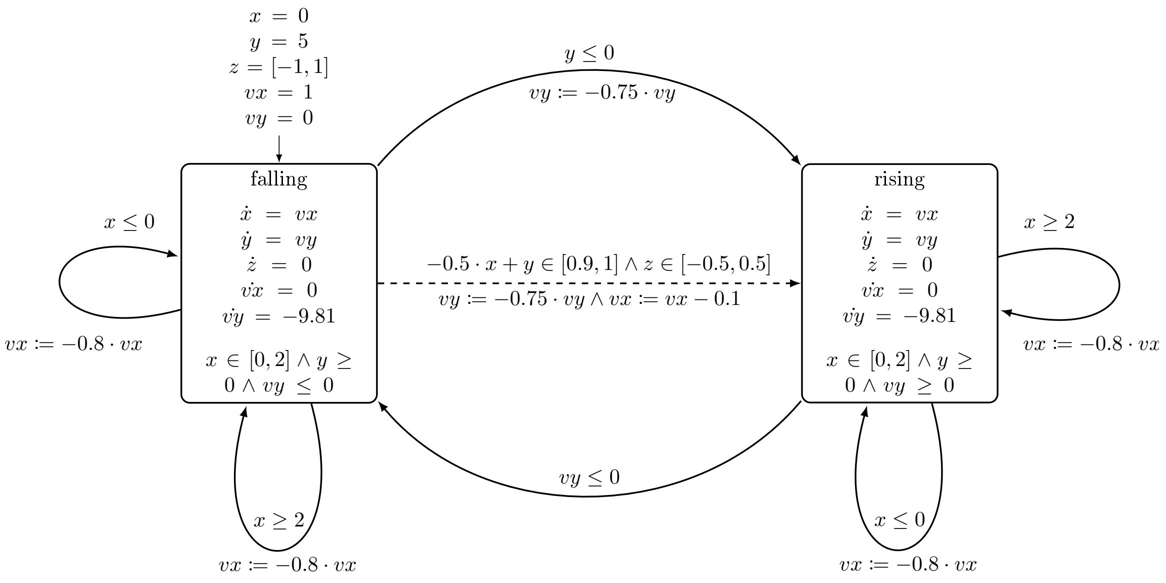


Figure A.4: Bouncing ball with tilted beam. Urgent edges are indicated by dashed lines.

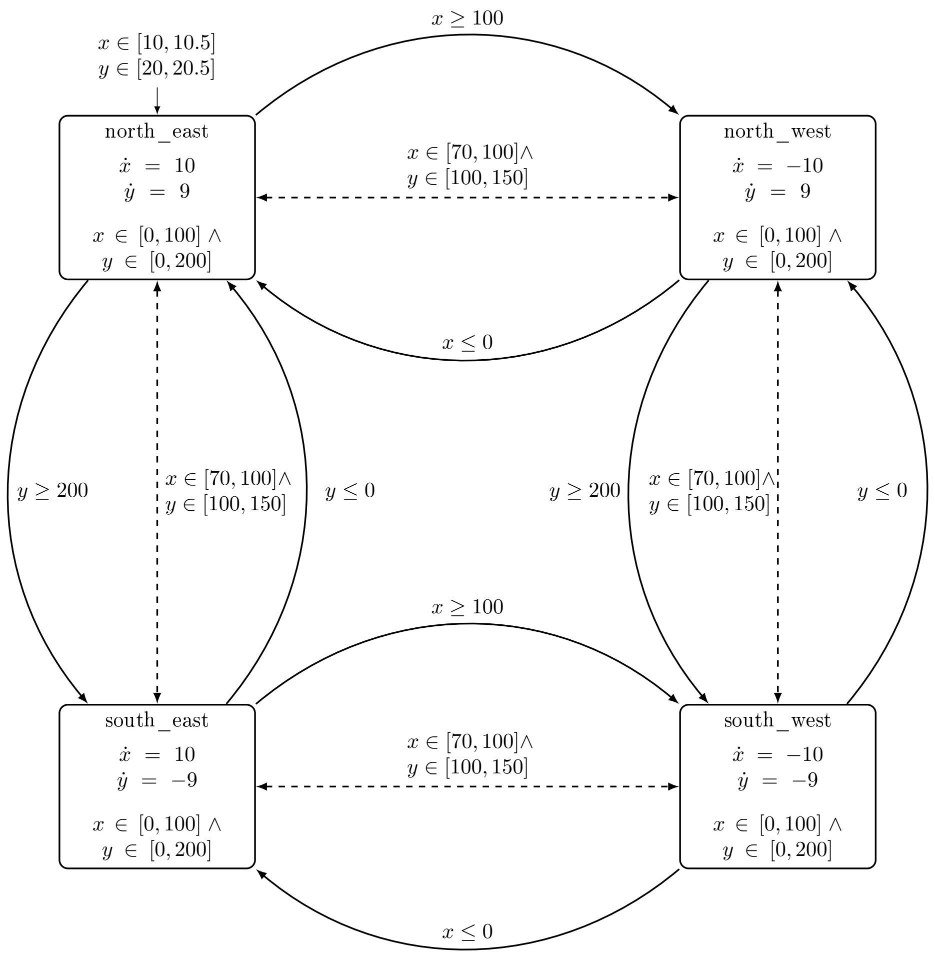


Figure A.5: Lawn mower with one unsafe zone. Urgent edges are indicated by dashed lines.

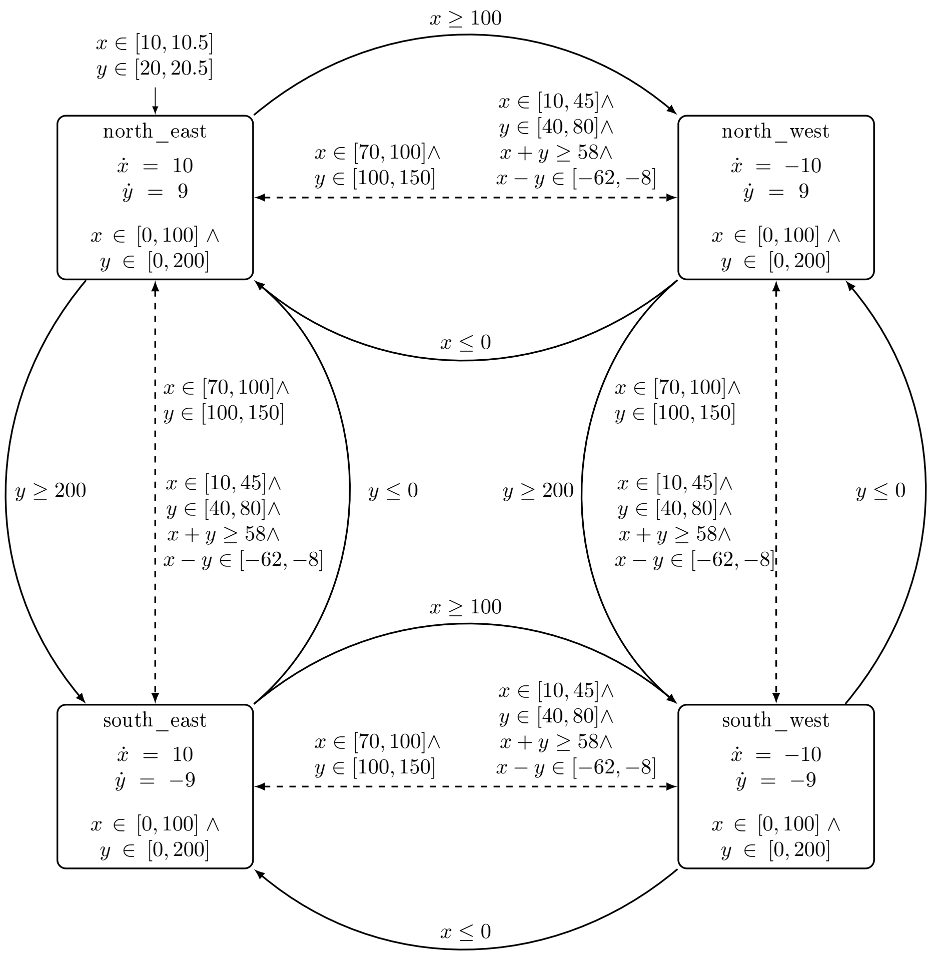


Figure A.6: Lawn mower with two unsafe zones. Urgent edges are indicated by dashed lines. Multiple guards on one line indicate separate transitions.

Appendix B

Benchmark Results

Table B.1: Verification times in seconds with box representation using exact (rational) arithmetic. Refinement levels are labeled U (unrefined), C (cut-off) and SD (set difference), heuristics are None (default strategy), Count, Vol. (Volume) and CC (Constraint-count). Timeout (TO) is 20 minutes.

Instance	Prune	Refinement levels and heuristic								
		SD	U, SD				U, C, SD			
			None	Count	Vol.	CC	None	Count	Vol.	CC
Vehicle	No	4.24	3.17	3.10	2.35	3.15	3.89	3.92	2.59	3.92
	Yes		3.02	3.07	2.22	3.00	3.89	3.90	2.33	3.94
Rod reactor	No	0.68	1.23	1.43	1.65	1.23	1.85	1.50	1.67	1.28
	Yes		1.39	1.62	1.83	1.39	2.00	1.68	1.85	1.44
BB hor.	No	3.44	4.28	4.30	4.36	4.32	6.71	6.93	6.98	6.98
	Yes		4.43	4.54	4.52	4.44	6.80	7.04	7.00	7.09
BB tilted	No	TO	6.39	6.43	6.36	6.34	11.66	11.93	11.98	11.85
	Yes		6.77	6.75	6.77	6.74	11.93	12.34	12.23	12.25
Lawn mower 1	No	4.28	10.39	3.92	43.74	10.46	4.13	4.10	45.81	11.04
	Yes		2.89	2.85	2.84	2.85	2.84	2.85	2.85	2.84
Lawn mower 2	No	272.24	191.73	113.18	TO	189.75	319.30	297.90	TO	438.16
	Yes		189.90	181.12	TO	191.28	324.11	301.57	852.33	429.17

Table B.2: Verification times in seconds with box representation using floating point arithmetic. Refinement levels are labeled U (unrefined), C (cut-off) and SD (set difference), heuristics are None (default strategy), Count, Vol. (Volume) and CC (Constraint-count). Timeout (TO) is 20 minutes.

Instance	Prune	Refinement levels and heuristic								
		SD	U, SD				U, C, SD			
			None	Count	Vol.	CC	None	Count	Vol.	CC
Vehicle	No	1.66	0.19	0.19	0.15	0.19	0.36	0.36	0.19	0.35
	Yes		0.23	0.23	0.17	0.23	0.42	0.41	0.18	0.42
Rod reactor	No	0.06	0.11	0.13	0.19	0.11	0.16	0.20	0.26	0.16
	Yes		0.11	0.14	0.20	0.11	0.17	0.20	0.26	0.17
BB hor.	No	0.61	0.28	0.28	0.28	0.28	0.51	0.51	0.51	0.51
	Yes		0.30	0.30	0.30	0.30	0.53	0.53	0.53	0.53
BB tilted	No	TO	0.66	0.66	0.67	0.66	1.34	1.34	1.35	1.34
	Yes		0.68	0.68	0.69	0.68	1.36	1.36	1.37	1.36
Lawn mower 1	No	1.19	0.87	0.31	16.19	0.86	1.11	0.36	18.15	1.11
	Yes		0.12	0.12	0.15	0.12	0.12	0.12	0.15	0.13
Lawn mower 2	No	170.21	124.32	63.47	993.40	117.71	189.36	108.47	552.88	247.22
	Yes		126.46	92.07	939.27	123.49	197.54	97.45	487.91	246.73

Table B.3: Verification times in seconds with $\mathcal{H}$-polytope representation using exact (rational) arithmetic. Refinement levels are labeled U (unrefined), C (cut-off) and SD (set difference), heuristics are None (default strategy), Count, Vol. (Volume) and CC (Constraint-count). Timeout (TO) is 20 minutes.

Instance	Prune	Refinement levels and heuristic								
		SD	U, SD				U, C, SD			
			None	Count	Vol.	CC	None	Count	Vol.	CC
Vehicle	No	TO	290.75	289.05	215.56	288.96	296.86	295.79	208.80	293.96
	Yes		264.60	264.24	181.85	266.54	254.55	255.15	180.07	254.63
Rod reactor	No	10.38	17.48	20.15	20.99	17.47	22.42	25.45	26.20	22.30
	Yes		20.03	22.60	23.77	19.92	24.98	28.15	29.01	24.49
BB hor.	No	TO	286.97	286.65	287.72	289.09	295.40	290.18	293.87	292.56
	Yes		288.35	291.01	292.05	288.85	292.83	292.12	296.70	294.83
BB tilted	No	TO	349.25	349.82	349.23	349.77	350.02	350.53	354.64	352.73
	Yes		350.67	350.53	352.27	352.34	351.33	352.81	356.25	359.10
Lawn mower 1	No	132.97	242.77	66.84	190.53	243.68	114.99	47.71	128.65	115.18
	Yes		73.35	74.09	73.55	73.72	74.36	73.85	74.02	73.62
Lawn mower 2	No	544.13	TO	408.31	TO	TO	751.88	461.01	TO	923.14
	Yes		447.78	388.50	569.73	407.74	412.64	372.91	586.96	363.58