Modeling and Analysis of Hybrid Systems

Lecture Notes

Erika Ábrahám and Stefan Schupp

Version: August 24, 2017

Theory of Hybrid Systems
Informatik 2

Faculty of Mathematics, Computer Science, and Natural Sciences
RWTH Aachen University

Contents

Basic Notations

We use the following standard notations:

Symbol	Description	
Sets and set operators		
$A \cup B$	$= \{x \mid x \in A \vee x \in B\}$	set union
$A \cap B$	$= \{x \mid x \in A \wedge x \in B\}$	set intersection
$A \backslash B$	$= \{x \mid x \in A \wedge x \notin B\}$	set minus
$A \times B$	$= \{(a, b) \mid a \in A \wedge b \in B\}$	cross product of two sets A and B
$\mathbb{Z}$		set of integers
$\mathbb{N}$		set of natural numbers including 0
$\mathbb{N}^d$	$= \underbrace{\mathbb{N} \times \ldots \times \mathbb{N}}_{d \text{ times}}$	the n-dimensional space of natural numbers
$\mathbb{N}_{>0}$	$= \mathbb{N} \backslash \{0\}$	set of positive natural numbers
$\mathbb{Q}$		set of rational numbers
$\mathbb{R}$		set of real numbers
$\mathbb{R}^d$	$= \underbrace{\mathbb{R} \times \ldots \times \mathbb{R}}_{d \text{ times}}$	the d-dimensional real space
$\mathbb{R}_{\geq 0}$	$= \{x \in \mathbb{R} \mid x \geq 0\}$	set of non-negative real numbers
$\mathbb{R}_{>0}$	$= \{x \in \mathbb{R} \mid x > 0\}$	set of positive real numbers
2^M	$= \{P \mid P \subseteq M\}$	powerset of the set M
Mappings		
id	$: M \to M$	identity mapping for a set M with $id(m) = m$ for all $m \in M$
$f(M)$	$= \{f(m) \mid m \in M\} \subseteq D_2$	image of a set $M \subseteq D_1$ according to a mapping $f : D_1 \to D_2$

Chapter 1

Introduction

Most areas of computer science deal with *discrete systems*, i.e., systems whose evolution can be described by a sequence of discrete state changes. Prominent examples are programs executing a sequence of computation steps, each of them possibly modifying the program's heap and stack. When we are only interested in the program's input-output behavior, such computation steps can be seen as instantaneous, *discrete* state changes.

Whereas the modeling and the analysis of discrete systems is a typical computer science subject, the development of methods and tools for the modeling and simulation of *dynamic systems* is hosted mainly in physics and control theory. Quantities of dynamic systems, like the temperature of a room or the speed of an object, evolve *continuously* over time according to the laws of physics in dependence on the current system state and the influence of the environment.

Discrete systems (e.g., sensors, chips, computer programs) are often used to control the behavior (e.g., temperature, speed, acceleration) of dynamic systems. The resulting system, consisting of the controller together with the controlled system, exhibits a *combined discrete and continuous behavior*, and is therefore called a *hybrid system*. Hybrid systems can have quite complex behavior, posing a challenging task for their analysis.

This book is devoted to modeling formalisms and algorithmic analysis techniques for different classes of hybrid systems, from the view point of computer science. It can be used as learning material for undergraduate courses, but also as a state-of-the-art overview for graduate students and researchers.

The contents of the book are as follows:

- In Chapter 2 we recall some automata-based *modeling* approaches for discrete systems, in the absence of dynamic behavior. We use *labeled state transition systems* (*Kripke structures*) to model finite-state systems, and *labeled transition systems* to model general discrete systems with possibly infinite state spaces. We define the *temporal logics* LTL, CTL, and CTL* to specify properties of discrete systems, and give a short introduction to (explicit) CTL *model checking* for labeled state transition systems. We also discuss *discrete-time systems* in a nutshell in Chapter 2.4 before we deal with continuous-time systems in the following chapters.
- We introduce *hybrid systems* and as a modeling language *hybrid automata* in Chapter 3. In the following chapters we consider different subclasses of hybrid automata with increasing expressive power, and methods for their safety analysis.
- *Timed automata* [AD94, BK08], extending discrete systems with a notion of time, are introduced in Chapter 4. We use the timed temporal logic TCTL to specify properties

of timed automata. We show that the validity of TCTL properties for timed automata is decidable by giving the standard model checking algorithm.

- Timed automata are quite restrictive in their modeling power. In Chapter 5 we define *rectangular automata*, a bit more general class, which is at the boundary of decidability: though checking TCTL properties of *initial* rectangular automata is a decidable problem, relaxing any of the restrictions on the expressivity of the modeling language leads to undecidability. We give the decidability proof following [HKPV98] in form of a reduction to TCTL model checking for timed automata.
- Even if the reachability problem for more expressive modeling formalisms for hybrid systems is in general undecidable, we need them to model more complex systems without too strong abstraction. Though undecidability implies that we cannot give any complete model checking algorithm for them, there might exist useful incomplete algorithms for their analysis. Such a more expressive model class is given by *linear hybrid automata I*[1], being the subject of Chapter 6. They are particularly interesting, because the bounded reachability problem (reachability within a fixed finite number of steps) is still decidable and efficiently computable for this class. We discuss a fixed-point-based algorithm from [ACH+95] and mention some approximation and abstraction techniques.
- To increase the expressive power of modeling, in Chapter 7 we introduce *linear hybrid automata II*. The dynamics in these models is specified by linear ordinary differential equations. The reachability analysis for hybrid automata requires special (over-approximative) representation techniques for state sets. We discuss representations by different geometric objects like convex polyhedra, oriented rectangular hulls, zonotopes, support functions and orthogonal polyhedra. Using such representations, we discuss an incomplete over-approximative fixed-point-based algorithm for reachability analysis.

Regarding undergraduate courses, the contents are determined such that they demonstrate the application and usefulness of a wide range of general computer science methods and techniques:

- *Formal modeling:* labeled state transition systems (Section 2.1.1), labeled transition systems (Section 2.1.2), discrete-time models (Chapter 2.4), timed automata (Section 4.1), rectangular automata (Section 5.1), linear hybrid automata (Section 6.1), general hybrid automata (Section 3.2);
- *Logics to formalize system properties:* propositional logic (Section 2.2.1), temporal logics LTL, CTL, CTL* (Section 2.2.2), timed temporal logic TCTL (Section 4.2);
- *Decidability issues:* proving decidability constructively by giving a finite bisimulation-based abstraction (Section 4.3 for timed automata), and proving decidability by reducing the question to a known problem (Section 5.2 for initialized rectangular automata).
- *Model checking:* CTL properties for labeled state transition systems (Section 2.3), TCTL properties for timed automata (Section 4.3), fixedpoint-based reachability analysis (Sections 6.2, 6.3 and **??**), minimization (Section 6.5);
- *Approximation:* for state sets of linear (Section 6.4) and general (Section **??**) hybrid automata.

[1] There are two different notions of linear hybrid automata. We mean here systems with a linear behavior, and not with linear differential equations describing the continuous behavior.

Chapter 2

Discrete Systems

In the next chapters we address the modeling and analysis of hybrid systems. Before doing so, in this chapter we first recall some fundamentals about the *modeling* of discrete systems in Section 2.1 and about *logics* that allow to formalize properties of discrete systems in Section 2.2. Finally, we explain the basic idea of (explicit) *CTL model checking* for discrete finite-state systems in Section 2.3.

2.1 Modeling Languages

As *modeling languages* we use in this chapter *labeled state transition systems* (generally known as *Kripke structures*, see Section 2.1.1) and *labeled transition systems* (Section 2.1.2) which additionally allow variables in the model.

2.1.1 Labeled State Transition Systems (LSTSs)

Labeled state transition systems consist of a set of states, a set of initial states where the execution starts, and labeled transitions between the states.

Definition 2.1 (Syntax of labeled state transition systems). *A* labeled state transition system (LSTS) *is a tuple* $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ *with* *LSTS* $\mathcal{LSTS}$

- *a (possibly infinite) set* Σ *of* states, Σ
- *a set* Lab *of (*synchronization*)* labels, Lab
- *a set* $Edge \subseteq \Sigma \times Lab \times \Sigma$ *of labeled* transitions *or* edges*, and* $Edge$
- *a non-empty set* $Init \subseteq \Sigma$ *of* initial states. $Init$

The semantics allows to build paths of an LSTS starting in an initial state and following transitions.

Definition 2.2 (Semantics of LSTS). *The* operational semantics *of a labeled state transition system* $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ *is given by the following single rule:*

$$\frac{(\sigma, a, \sigma') \in Edge}{\sigma \xrightarrow{a} \sigma'} \; Rule_{discrete}$$

We call $\sigma \xrightarrow{a} \sigma'$ *an* (execution) step. *A* path *(or* run *or* execution*)* π *of* $\mathcal{LSTS}$ *is a (finite or infinite) sequence* $\pi = \sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \ldots$. π

For a path $\pi = \sigma_0 \to \sigma_1 \to \ldots$ of $\mathcal{LSTS}$ and some $i \in \mathbb{N}$, $i \leq |\pi|$, let $\pi(i) = \sigma_i$ and $\pi^i = \sigma_i \to \sigma_{i+1} \to \ldots$.

$\Pi(\sigma)$ *We use $\Pi_{\mathcal{LSTS}}$ (or simply Π) to denote the set of all paths of $\mathcal{LSTS}$ and define $\Pi_{\mathcal{LSTS}}(\sigma) = \{\pi \in \Pi_{\mathcal{LSTS}} \mid \pi(0) = \sigma\}$.*
The path π is initial *if $\pi(0)$ is an initial state. A state is* reachable *iff there is an initial path leading to it.*

We sometimes simply write $\pi = \sigma_0 \to \sigma_1 \to \ldots$ when the labels of the edges are not of interest. We say that $\sigma' \in \Sigma$ is a *successor* of $\sigma \in \Sigma$ and σ is a *predecessor* of σ' iff $\sigma \xrightarrow{a} \sigma'$ for some $a \in Lab$. Note that for a path $\pi = \sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \ldots$ the state σ_{i+1} is a successor of σ_i for each $0 \leq i < |\pi|$, where $|\pi|$ denotes the number of steps in the path (possibly being infinity).

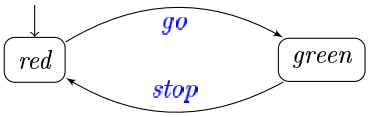
The labels of the set *Lab* are attached to edges and are used for synchronization purposes in the parallel composition (see page 10).

To be able to formalize properties of LSTSs, it is common to define a set of *atomic propositions*

AP, L AP and a labeling function $L : \Sigma \to 2^{AP}$ assigning a set of atomic propositions to each state. The set $L(\sigma) \subseteq AP$ consists of all propositions that are defined to hold in the state σ. These propositional labels on states should not be mixed up with the synchronization labels on edges.

A labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ can be represented as a directed graph, where the vertices of the graph are the states from Σ and the (labeled) edges are the transitions from *Edge*. The initial states are marked by an incoming edge without source.

Example 2.1 (Pedestrian light). *We model a pedestrian traffic light in a crossing by a labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$. The light can be red or green (we do not model the light being off or blinking). Thus we can represent the light's state set by $\Sigma = \{red, green\}$. Assume the light is initially red, i.e., $Init = \{red\}$. Possible state changes go from red to green and from green to red, yielding $Edge = \{(red, go, green), (green, stop, red)\}$ for a possible label set $Lab = \{go, stop\}$. The labels can be used, e.g., to synchronize state changes with another light in the same crossing. The model $\mathcal{LSTS}$ can be visualized as follows:*



This model is deterministic*, i.e., both of its states has a single possible successor state. The system has the single initial run $red \xrightarrow{go} green \xrightarrow{stop} red \ldots$.*

Larger or more complex systems are often modeled componentwise, such that the whole system is given by the *parallel composition* of the components. Component-local, non-synchronizing transitions, having labels belonging to one component's label set only, are executed in an interleaved manner. Synchronizing transitions of the components, agreeing on the label, are executed synchronously.

Definition 2.3 (Parallel composition of LSTS). *Let*

$$\begin{aligned} \mathcal{LSTS}_1 &= (\Sigma_1, Lab_1, Edge_1, Init_1) \text{ and} \\ \mathcal{LSTS}_2 &= (\Sigma_2, Lab_2, Edge_2, Init_2) \end{aligned}$$

$\mathcal{LSTS}_1 || \mathcal{LSTS}_2$ *be two LSTSs. The* parallel composition $\mathcal{LSTS}_1 || \mathcal{LSTS}_2 = (\Sigma, Lab, Edge, Init)$ *is an LSTS with*

- $\Sigma = \Sigma_1 \times \Sigma_2$,
- $Lab = Lab_1 \cup Lab_2$,
- $((s_1, s_2), a, (s_1', s_2')) \in Edge$ *iff*
  1. $a \in Lab_1 \cap Lab_2$, $(s_1, a, s_1') \in Edge_1$, *and* $(s_2, a, s_2') \in Edge_2$, *or*
  2. $a \in Lab_1 \backslash Lab_2$, $(s_1, a, s_1') \in Edge_1$, *and* $s_2 = s_2'$, *or*
  3. $a \in Lab_2 \backslash Lab_1$, $(s_2, a, s_2') \in Edge_2$, *and* $s_1 = s_1'$,
- $Init = Init_1 \times Init_2$.

To demonstrate the advantages of compositional modeling, we give an example for the parallel composition of two traffic lights.

Example 2.2 (Two pedestrian lights). *Assume now a crossing of two roads with two pedestrian lights, similar to those from Example 2.1, one in north-south and one in east-west direction. The two lights are composed such that they allow pedestrians to pass alternatingly.*

Formally, the two LSTSs are given by

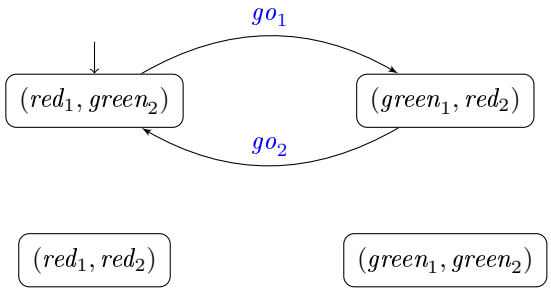
$$\mathcal{LSTS}_1 = (\underbrace{\{red_1, green_1\}}_{\Sigma_1}, \underbrace{\{go_1, go_2\}}_{Lab_1}, \underbrace{\{(red_1, go_1, green_1), (green_1, go_2, red_1)\}}_{Edge_1}, \underbrace{\{red_1\}}_{Init_1})$$

$$\mathcal{LSTS}_2 = (\underbrace{\{red_2, green_2\}}_{\Sigma_2}, \underbrace{\{go_1, go_2\}}_{Lab_2}, \underbrace{\{(red_2, go_2, green_2), (green_2, go_1, red_2)\}}_{Edge_2}, \underbrace{\{green_2\}}_{Init_2})$$

The parallel composition $\mathcal{LSTS}_1 || \mathcal{LSTS}_2 = (\Sigma, Lab, Edge, Init)$ *is by definition:*

$$\begin{aligned} \Sigma &= \{(green_1, green_2), (green_1, red_2), (red_1, green_2), (red_1, red_2)\} \\ Lab &= \{go_1, go_2\} \\ Edge &= \{((red_1, green_2), go_1, (green_1, red_2)), ((green_1, red_2), go_2, (red_1, green_2))\} \\ Init &= \{(red_1, green_2)\} \end{aligned}$$

The parallel composition can be visualized be the following graph:



Note that the states (red_1, red_2) *and* $(green_1, green_2)$ *are not reachable, i.e., the two lights are never green respectively red at the same time.*

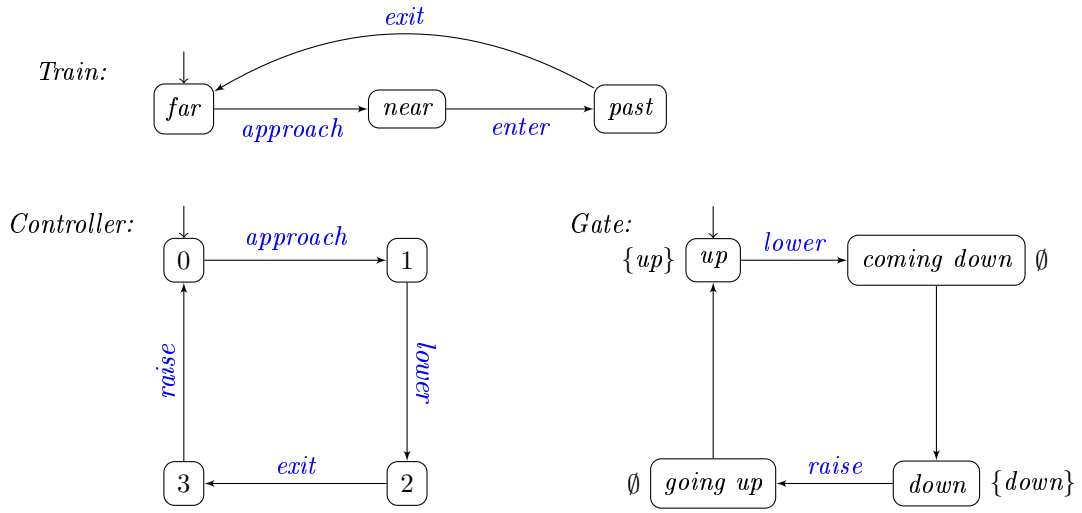
Again, the composition is deterministic and has a single initial run $(red_1, green_2) \overset{go_1}{\to} (green_1, red_2) \overset{go_2}{\to} (red_1, green2) \ldots$.

Another well-known example for the parallel composition is that of a railway crossing.

Example 2.3 (Railroad crossing). *Assume the crossing of a railroad with a street, secured by a gate. The system consists of three components: a train, a controller and a gate. The train communicates with the controller, and the controller communicates with the gate as follows.*

- *Sensors recognize when the train is approaching to the gate and an "approach" signal is sent to the controller. Similarly, if the train has left the railroad crossing, an "exit" signal gets sent to the controller.*
- *The controller reacts to an incoming "approach" signal with the sending of a "lower" signal to the gate. Similarly, when the controller receives an "exit" signal, it sends a "raise" signal to the gate.*
- *The gate reacts to an incoming "lower" signal with closing the gate, and to a "raise" signal with opening the gate.*

If we are interested in the communication aspects only, we can model the railroad crossing system as the parallel composition of the following three LSTS components:



Given the proposition set $AP = \{up, down\}$, *we can define a state labeling function* L *assigning a set of propositions to the states of the gate as depicted in the above graph.*

The formal specification of the parallel composition is the content of Exercise **??**. *The parallel composition's initial state is* $(far, 0, up)$. *In the initial state the gate cannot execute, because the only possible transition from the state up has the label lower that synchronizes with the controller, but the controller first has to move to the state* 1 *to be able to synchronize on it. Thus first synchronization on approach must take place. Therefore, each initial path of the composition starts with the step* $(far, 0, up) \overset{approach}{\to} (near, 1, up)$.

2.1.2 Labeled Transition Systems (LTSs)

Another, more expressive but still discrete modeling language are *labeled transition systems* (*LTSs*), which additionally allow *variables* in the model. Here we consider real-valued variables only, and in the following we restrict the formalisms accordingly.

Given a set of real-valued variables *Var*, a *valuation* is a function $\nu : Var \to \mathbb{R}$ assigning values to the variables. We use V_{Var} (or short V) to denote the set of all valuations for the variable set *Var*. *Var*, ν, V

An LTS has a finite set of *locations*, also called *modes*, which can possibly be entered with different valuations. The current *state* $\sigma = (\ell, \nu)$ of an LTS is determined by the current location l and the current valuation ν. A set of *initial states* specifies the states in which the execution may start. *Loc* σ, Σ

The locations of an LTS are connected by labeled *transitions* (edges). In contrast to LSTSs, each edge of an LTS can have a guard and an effect, specified in form of a *transition relation* $\mu \subseteq V \times V$: the transition can be taken with a valuation ν thereby changing the valuation to ν' iff $(\nu, \nu') \in \mu$.

Example 2.4. *Assume a variable set* $Var = \{x\}$ *and a transition that is enabled if* $x > 0$ *holds and it decreases the value of* x *by* 1. *The corresponding transition relation would be* $\mu = \{(\nu, \nu') \in V^2 \mid \nu(x) > 0 \wedge \nu'(x) = \nu(x) - 1\}$.

In the following definition of LTSs we also embed *controlled variables* and *τ-transitions* (also called *stutter* transitions). Their role will become clear later when we define the parallel composition of LTSs. Intuitively, these constructs help us to define "local", "output" or "write" variables of an LTS whose values may not be changed by non-synchronizing steps of other parallel LTSs.

Definition 2.4 (Syntax of labeled transition systems). *A* labeled transition system (LTS) *is a tuple* $\mathcal{LTS} = (Loc, Var, Con, Lab, Edge, Init)$ *with* *LTS*, $\mathcal{LTS}$

- *a finite set Loc of* locations,
- *a finite ordered set* $Var = \{x_1, \ldots, x_d\}$ *of real-valued* variables,
- *a function* $Con : Loc \to 2^{Var}$ *assigning a set of* controlled variables *to each location,* *Con*
- *a finite set Lab of labels, including the* stutter label $\tau \in Lab$, $\tau \in Lab$
- *a finite set* $Edge \subseteq Loc \times Lab \times 2^{V^2} \times Loc$ *of* edges *or* transitions *including a τ-transition* (ℓ, τ, Id, ℓ) *for each location* $\ell \in Loc$ *with* $Id = \{(\nu, \nu') \in V_{Var}^2 \mid \forall x \in Con(\ell).\ \nu'(x) = \nu(x)\}$, *and where all edges with label τ are τ-transitions, and* *Edge*, *Id*
- *a set* $Init \subseteq \Sigma$ *of* initial states,

where $\Sigma = Loc \times V_{Var}$ *denotes the state space of* $\mathcal{LTS}$.

Definition 2.5 (Semantics of LTSs). *The* operational semantics *of a labeled transition system* $\mathcal{LTS} = (Loc, Var, Con, Lab, Edge, Init)$ *is specified by the following single rule:*

$$\frac{(\ell, a, \mu, \ell') \in Edge \quad (\nu, \nu') \in \mu}{(\ell, \nu) \xrightarrow{a} (\ell', \nu')}\ Rule_{discrete}$$

We call $\sigma \xrightarrow{a} \sigma'$ *an* (execution) step, *which we also write as* $\sigma \to \sigma'$ *when we are not interested in its label. A* path *(or* run *or* execution*)* π *of* $\mathcal{LTS}$ *is a (finite or infinite) sequence* $\pi = \sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \ldots$; *for* $i \in \mathbb{N}$, $i \leq |\pi|$, *we define* $\pi(i) = \sigma_i$ *and* $\pi^i = \sigma_i \to \sigma_{i+1} \to \ldots$. $\pi, \pi(i), \pi^i$
We use $\Pi_{\mathcal{LTS}}$ *(or simply* Π*) to denote the set of all paths of* $\mathcal{LTS}$ *and define* $\Pi_{\mathcal{LTS}}(\sigma) = \{\pi \in \Pi_{\mathcal{LTS}} \mid \pi(0) = \sigma\}$. $\Pi(\sigma), \Pi$

```
   method mult(int y, int z){
     int x;
ℓ0   x := 0;
ℓ1   while( y > 0 ) {
ℓ2      y := y-1;
ℓ3      x := x+z;
     }
ℓ4 }
```

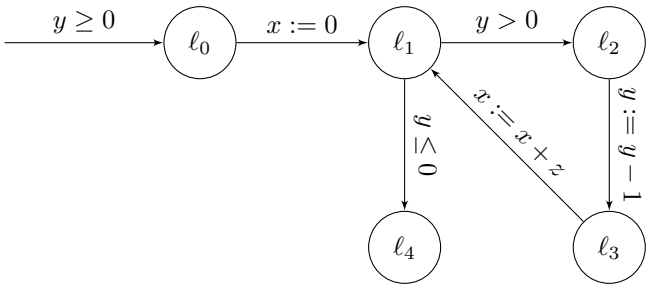


Figure 2.1: Modeling a simple while program with an LTS.

The path π is initial *if $\pi(0)$ is an initial state. A state is* reachable *iff there is an initial path leading to it.*

We sometimes simply write $\pi = \sigma_0 \to \sigma_1 \to \ldots$ when the labels of the edges are not of interest. Note again that for a path $\pi = \sigma_0 \overset{a_0}{\to} \sigma_1 \overset{a_1}{\to} \ldots$ there is a transition $\sigma_i \overset{a_i}{\to} \sigma_{i+1}$ between all successive states σ_i and σ_{i+1}, $0 \leq i < |\pi|$, in the path, where $|\pi|$ denotes the number of steps in the path (possibly being infinity).

Based on the operational semantics, an LTS induces an underlying LSTS state space model: a transition $(\ell, \nu) \overset{a}{\to} (\ell', \nu')$ can be performed in the induced LSTS if there is an edge (ℓ, a, μ, ℓ') from ℓ to ℓ' in the LTS with $(\nu, \nu') \in \mu$.

The *guard* $\{\nu \in V \mid \exists \nu' \in V.(\nu, \nu') \in \mu\}$ of a transition $(l, a, \mu, l') \in Edge$ specifies the set of valuations from which the transition can be taken. Transition guards are often specified by formulae of the first-order logic over the reals (without quantifiers). E.g., $x_i > 0$ specifies the guard $\{\nu \in V | \nu(x_i) > 0\}$. Transition *resets* are specified by the second component in the transition relation. Resets are often specified by Boolean combinations of assignments $x_i := e_i$, where x_i is a variable and e_i is an expression over the variables. E.g., the semantics of $x_i := e_i$ under a valuation ν is given by $\{\nu' \in V \mid \nu'(x_i) = \nu(e_i)\}$.

The next example shows how LTSs can be used to describe program execution.

Example 2.5 (Modeling a simple while program). *The simple while program on the left of Figure 2.1 calculates $x := y \cdot z$ for two input integers y and z with $y \geq 0$. Each instruction corresponds to a transition (ℓ, a, μ, ℓ') with the source location l being the program location before the instruction, the target location l' being the program location after the instruction, a label a which is omitted here because no synchronization is needed, and a set of valuation pairs μ describing the condition or effect represented by the instruction.*

Formally, this (closed) system can be defined as a transition system $\mathcal{LTS} = (Loc, Var, Con, Lab, Edge, Init)$ where

- $Loc = \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\}$,
- $Var = \{x, y, z\}$,
- $V = \{\nu \mid \nu : Var \to \mathbb{R}\}$ *and* $\Sigma = Loc \times V$,
- $Con(\ell) = Var$ *for each* $\ell \in Loc$,
- $Lab = \{a, \tau\}$,
- $Edge =$

$$\begin{array}{llll}
\{(\ell_0, & a, & \{(\nu,\nu') \in V^2 \mid \nu'(x) = 0 \wedge \nu'(y) = \nu(y) \wedge \nu'(z) = \nu(z)\}, & \ell_1), \\
(\ell_1, & a, & \{(\nu,\nu') \in V^2 \mid \nu(y) > 0 \wedge \nu' = \nu\}, & \ell_2), \\
(\ell_2, & a, & \{(\nu,\nu') \in V^2 \mid \nu'(x) = \nu(x) \wedge \nu'(y) = \nu(y) - 1 \wedge \nu'(z) = \nu(z)\}, & \ell_3), \\
(\ell_3, & a, & \{(\nu,\nu') \in V^2 \mid \nu'(x) = \nu(x) + \nu(z) \wedge \nu'(y) = \nu(y) \wedge \nu'(z) = \nu(z)\}, & \ell_1), \\
(\ell_1, & a, & \{(\nu,\nu') \in V^2 \mid \nu(y) \leq 0 \wedge \nu' = \nu\}, & \ell_4), \\
\multicolumn{4}{l}{\quad \tau_{\ell_0}, \tau_{\ell_1}, \tau_{\ell_2}, \tau_{\ell_3}, \tau_{\ell_4}\},}
\end{array}$$

- $Init = \{(\ell_0, \nu) \in \Sigma \mid \nu(y) \in \mathbb{N} \wedge \nu(z) \in \mathbb{Z}\}$

with $\tau_{\ell_i} = (\ell_i, \tau, \{(\nu,\nu') \in V^2 \mid \nu = \nu'\}, \ell_i)$ *for all* $l \in Loc$. *This LTS model is illustrated on the right of Figure 2.1 (without showing the transition label* a *and the* τ*-transitions).*

The *parallel composition* of LTSs allows to model larger systems compositionally. Intuitively, two LTSs running in parallel may execute non-synchronizing steps interleaved on their own, whereas synchronizing steps are executed simultaneously in both components. Whether a step is synchronizing or not depends on the fact whether both systems have the step's label in their label sets. One of the components can take a transition with a common label only if the other component also takes a transition with the same label. For this joint step the conditions and effects of both transitions must be considered, i.e., the transition relation for the joint step is the intersection of the transition relations of both local transitions.

If one of the components executes a local, non-synchronizing step, the other component is basically not active. However, in the parallel composition of LTSs we define the other component to take a so-called τ*-transition* or *stutter transition*, a "do nothing" step. The reason for this is twofold: Firstly, this makes the definitions and the underlying algorithms more unique, since in each step both systems take a transition. Secondly, and more importantly, sometimes we would like to define components with variables local to this component, or with variables that can only be read but not written by the other components. Then the τ-transitions of this component will specify in their transition relation that the values of those variables are not modified by the environment's non-synchronizing steps. Variables that a component has under its control and that must not be modified by the local steps of its environment are defined by the function *Con*.

Definition 2.6 (Parallel composition of LTSs). *Let*

$$\begin{array}{rcl}
\mathcal{LTS}_1 & = & (Loc_1, Var, Con_1, Lab_1, Edge_1, Init_1) \text{ and} \\
\mathcal{LTS}_2 & = & (Loc_2, Var, Con_2, Lab_2, Edge_2, Init_2)
\end{array}$$

be two LTSs. The parallel composition *or* product

$$\mathcal{LTS} = \mathcal{LTS}_1 \mid\mid \mathcal{LTS}_2 = (Loc, Var, Con, Lab, Edge, Init) \qquad \begin{array}{r}\mathcal{LTS}_1 \mid\mid \\ \mathcal{LTS}_2\end{array}$$

of $\mathcal{LTS}_1$ *and* $\mathcal{LTS}_2$ *is the LTS defined by*

- $Loc = Loc_1 \times Loc_2$,
- $Con((\ell_1, \ell_2)) = Con_1(\ell_1) \cup Con_2(\ell_2)$,
- $Lab = Lab_1 \cup Lab_2$,
- $((\ell_1, \ell_2), a, \mu, (\ell_1', \ell_2')) \in Edge$ *iff*
  - *there exist* $(\ell_1, a_1, \mu_1, \ell_1') \in Edge_1$ *and* $(\ell_2, a_2, \mu_2, \ell_2') \in Edge_2$ *such that*
  - *either* $a_1 = a_2 = a$ *or*
    $a_1 = a \in Lab_1 \backslash Lab_2$ *and* $a_2 = \tau$, *or*
    $a_1 = \tau$ *and* $a_2 = a \in Lab_2 \backslash Lab_1$, *and*
  - $\mu = \mu_1 \cap \mu_2$, *and*
- $Init = \{((\ell_1, \ell_2), \nu) \mid (\ell_1, \nu) \in Init_1 \wedge (\ell_2, \nu) \in Init_2\}$.

Example 2.6. *Assume the parallel composition of the following two LTSs:*

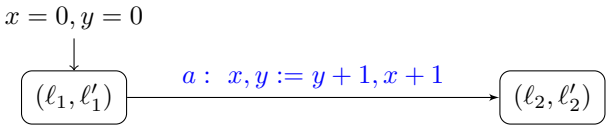
$$\begin{aligned} \mathcal{LTS}_1 &= (Loc_1, Var, Con_1, Lab_1, Edge_1, Init_1) \\ \mathcal{LTS}_2 &= (Loc_2, Var, Con_2, Lab_2, Edge_2, Init_2) \end{aligned}$$

with

- $Loc_1 = \{\ell_1, \ell_2\}$, $Loc_2 = \{\ell'_1, \ell'_2\}$,
- $Var = \{x, y\}$,
- $Con_1(\ell_1) = Con_1(\ell_2) = \{x\}$, $Con_2(\ell'_1) = Con_2(\ell'_2) = \{y\}$,
- $Lab_1 = Lab_2 = \{a, \tau\}$,
- $Edge_1 = \{(\ell_1,\ a,\ \{(\nu, \nu') \in V^2 \mid \nu'(x) = \nu(y) + 1\},\ \ell_2),\ \tau_{\ell_1},\ \tau_{\ell_2}\}$,
  $Edge_2 = \{(\ell'_1,\ a,\ \{(\nu, \nu') \in V^2 \mid \nu'(y) = \nu(x) + 1\},\ \ell'_2),\ \tau_{\ell'_1},\ \tau_{\ell'_2}\}$,
- $Init_1 = \{(\ell_1, \{\nu \in V | \nu(x) = 0\})\}$, $Init_2 = \{(\ell'_1, \{\nu \in V | \nu(y) = 0\})\}$

with $\tau_\ell = \{(\nu, \nu') \in V^2 \mid \forall v \in Con_i(\ell).\ \nu(v) = \nu'(v)\}$ *for all* $i = 1, 2$ *and* $\ell \in Loc_i$. *Graphically (without representing the control variables and* τ*-transitions):*

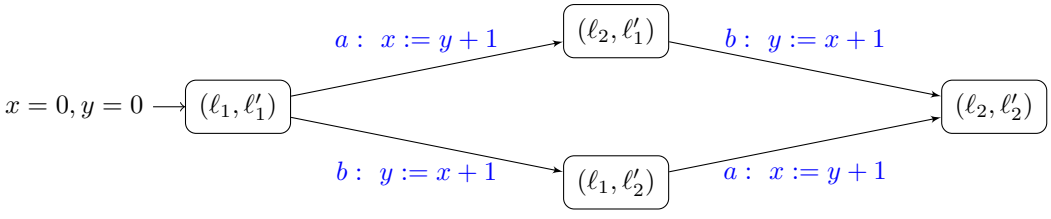
The graphical representation of the parallel composition (again without control variables and τ*-transitions) looks as follows:*



As the only non-τ*-transitions of the LTSs are synchronized by the label* a*, all runs of the system are of the form* $\sigma_0 \xrightarrow{\tau} \ldots \sigma_0 \xrightarrow{a} \sigma_1 \xrightarrow{\tau} \sigma_1 \ldots$ *with* $\sigma_0(x) = \sigma_0(y) = 0$ *and* $\sigma_1(x) = \sigma_1(y) = 1$.

Let us modify the example such that the transitions do not synchronize:

The parallel composition looks as follows:



Now the transitions interleave, and if we skip the transitions where both components do a τ*-step, we get two possible runs:*

- $\sigma_0 \xrightarrow{a} \sigma_1 \xrightarrow{b} \sigma_2$ *with*
  - $\sigma_0 = ((\ell_1, \ell'_1), \nu_0)$, $\nu_0(x) = \nu_0(y) = 0$,
  - $\sigma_1 = ((\ell_2, \ell'_1), \nu_1)$, $\nu_1(x) = 1$, $\nu_1(y) = 0$, *and*
  - $\sigma_2 = ((\ell_2, \ell'_2), \nu_2)$, $\nu_2(x) = 1$, $\nu_2(y) = 2$, *or*
- $\sigma_0 \xrightarrow{b} \sigma_1 \xrightarrow{a} \sigma_2$ *with*
  - $\sigma_0 = ((\ell_1, \ell'_1), \nu_0)$, $\nu_0(x) = \nu_0(y) = 0$,
  - $\sigma_1 = ((\ell_1, \ell'_2), \nu_1)$, $\nu_1(x) = 0$, $\nu_1(y) = 1$, *and*
  - $\sigma_2 = ((\ell_2, \ell'_2), \nu_2)$, $\nu_2(x) = 2$, $\nu_2(y) = 1$.

2.2 Temporal Logics

For the formalization of properties for discrete systems, after introducing *propositional logic* in a nutshell in Section 2.2.1 we deal with the temporal logics LTL, CTL and CTL* in Section 2.2.2.

2.2.1 Propositional Logic

Assume a set of states Σ, a set of atomic propositions AP, and a labeling function $L : \Sigma \to 2^{AP}$ assigning to each state a set of propositions holding in that state. Then we can use *propositional logic* to describe properties of states. Propositional logic formulae are built from atomic propositions using Boolean operators according to the following abstract syntax:

$$\varphi \ ::= \ a \mid (\varphi \wedge \varphi) \mid (\neg\varphi) \qquad \wedge, \neg$$

with $a \in AP$ and where $\wedge$ is the "and"-operator for conjunction and $\neg$ is the operator for negation. As syntactic sugar the constants *true* and *false*, and further Boolean operators like $\vee$ ("or"), $\to$ ("implies"), $\leftrightarrow$ ("if and only if"), etc. can be introduced. We often omit parentheses with the convention that the strength of binding is in the order $\neg, \wedge, \vee, \to, \leftrightarrow$, i.e., $\neg$ binds the strongest and $\leftrightarrow$ the weakest. We use $Form^{AP}_{prop}$ (or short $Form_{prop}$) to denote the set of all propositional logic formulae over the atomic proposition set AP. $\vee, \to, \leftrightarrow$ $Form_{prop}$

Propositional logic formulae are evaluated in the context of a state with the help of the labeling function. The semantics is given by the relation $\models_{prop} \subseteq \Sigma \times Form_{prop}$ (or short $\models$), which is defined recursively over the structure of propositional logic formulae as follows: $\models_{prop}$

$$\begin{array}{lll} \sigma \models_{prop} a & \textit{iff} & a \in L(\sigma), \\ \sigma \models_{prop} (\varphi_1 \wedge \varphi_2) & \textit{iff} & \sigma \models_{prop} \varphi_1 \textit{ and } \sigma \models_{prop} \varphi_2, \\ \sigma \models_{prop} (\neg\varphi) & \textit{iff} & \sigma \not\models_{prop} \varphi. \end{array}$$

Though propositional logic is well-suited to describe *states* of a system, we are also interested in describing *computations* of systems. Propositional logic extended with temporal modalities can be used for this purpose.

2.2.2 Temporal Logics

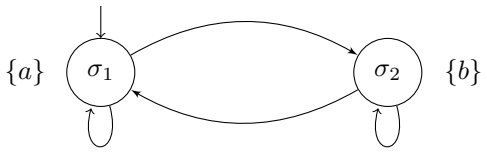
Assume in the following a labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$, a set of atomic propositions AP, and a labeling function $L : \Sigma \to 2^{AP}$. The semantics of $\mathcal{LSTS}$ specifies its behavior as a set of paths. This path set can also be seen as a set of trees, which we get by sharing common prefixes of paths and branching only at the place of the first difference. I.e., for

each initial state there is a *computation tree* and each path of the system corresponds to a path in one of the trees.

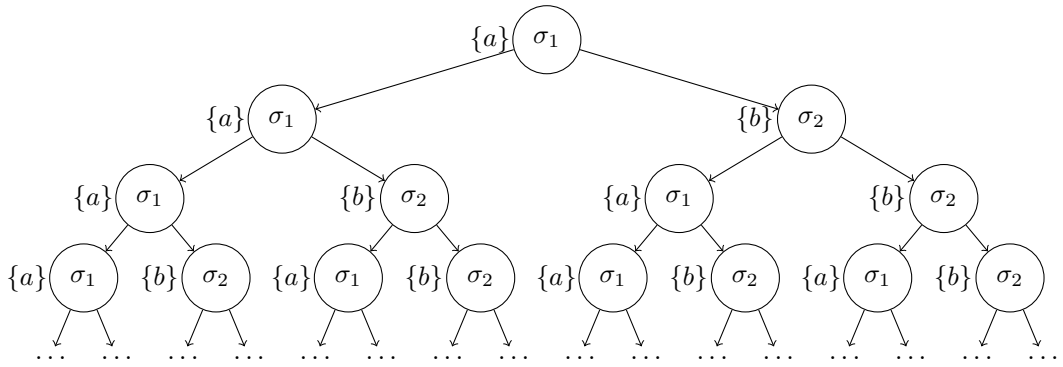
For deterministic systems with a single initial state, the computation tree is just a line. For non-deterministic systems, the branching at a node of a computation tree represents possible non-deterministic choices for further execution. Each reachable state is represented as a node in one of the trees as many times as the number of different (finite) paths leading to it. Note that this might happen infinitely often when the state is part of a reachable loop.

In the following we assume deadlock-free systems, i.e., infinite computation trees.

Example 2.7 (Computation tree). *Assume the following simple state transition system, where we omit synchronization labels on edges, but depict the labeling of states with atomic propositions:*



This system has the following computation tree:



Next we describe the temporal logics LTL, CTL, and CTL*, which are suited to argue about paths in the computation tree.

Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is suited to argue about single (linear) paths in the computation tree.

Definition 2.7 (Syntax of LTL). *Assume a set AP of atomic propositions. LTL has the abstract* syntax

$\mathcal{X}$,$\mathcal{U}$

$$\varphi \quad ::= \quad a \mid (\varphi \wedge \varphi) \mid (\neg\varphi) \mid (\mathcal{X}\varphi) \mid (\varphi \,\mathcal{U}\, \varphi)$$

$Form_{LTL}$ *where $a \in AP$. We use $Form_{LTL}^{AP}$ (or short $Form_{LTL}$) to denote the set of LTL formulae over AP.*

Again, we omit parentheses when it causes no confusion, assuming that the Boolean operators bind stronger than the temporal ones.

Recall that for a path $\pi = \sigma_0 \to \sigma_1 \to \ldots$ of $\mathcal{LSTS}$ and some $i \in \mathbb{N}$, $i \leq |\pi|$, we defined $\pi(i) = \sigma_i$ and $\pi^i = \sigma_i \to \sigma_{i+1} \to \ldots$ (see Definition 2.2).

A path π satisfies a proposition $a \in AP$ if the proposition holds in the first state $\pi(0)$ of π, i.e., if $a \in L(\pi(0))$. Using the "next time" temporal operator $\mathcal{X}$ we can build LTL formulae $\mathcal{X}\varphi$ ("next time φ") which are satisfied by a path π iff φ holds in π^1, i.e., when removing the first state from π. The second temporal operator is the "until" operator. The formula $\varphi_1 \mathcal{U}\ \varphi_2$ ("φ_1 until φ_2") is satisfied by a path $\pi = \sigma_0 \to \sigma_1 \to \ldots$ iff φ_2 holds for some suffix π^j and φ_1 holds all the time before, i.e., for all π^i with $0 \leq i < j$.

As syntactic sugar the temporal operators $\mathcal{F}$ ("finally" or "eventually") and $\mathcal{G}$ ("globally") can be introduced. The formula $\mathcal{F}\varphi$ ("finally φ) is defined as $true\mathcal{U}\,\varphi$, stating that φ will be true after a finite number of steps. The formula $\mathcal{G}\varphi$ ("globally φ") is defined as $\neg(true\,\mathcal{U}\,\neg\varphi)$, stating that φ holds all along the path. $\mathcal{F}, \mathcal{G}$

Remark 2.1. Some approaches define two further temporal operators which we do not use in the following but mention them for completeness. The first one is the "release" operator: $\varphi_1 \mathcal{R}\,\varphi_2$, defined as $\neg((\neg\varphi_1)\,\mathcal{U}\,(\neg\varphi_2))$, expresses that φ_2 holds either forever or until $\varphi_1 \wedge \varphi_2$ gets valid. The second one is the "weak until": the formula $\varphi_1 \mathcal{U}_{weak}\,\varphi_2$, defined as $(\varphi_1\,\mathcal{U}\,\varphi_2) \vee (\mathcal{G}\varphi_1)$, weakens the meaning of "until" with the possibility that φ_1 holds forever without φ_2 becoming true.

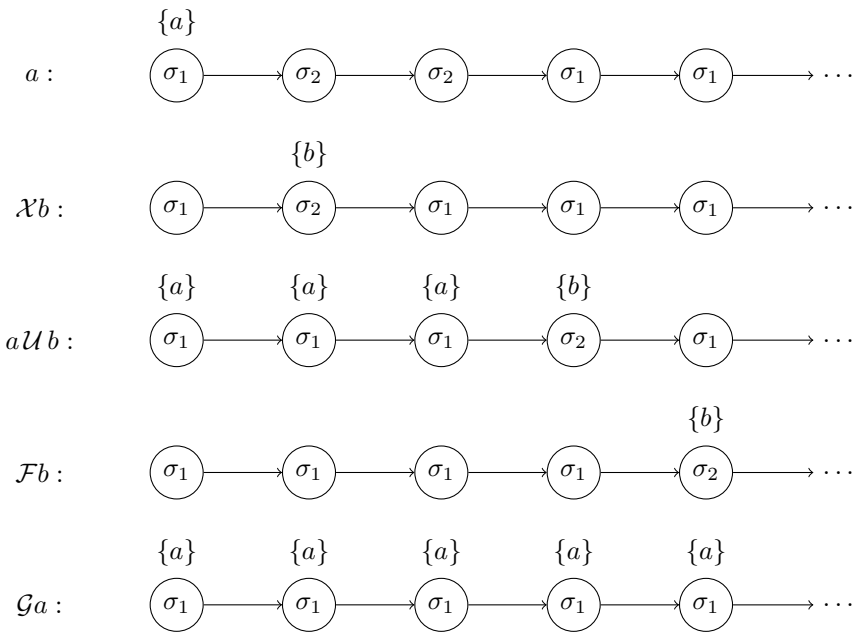
It is also possible to define operators "since", "previous", "once", "always been" referring to the past. They are symmetric to "until", "next", "finally" and "globally", but they refer to the past computation instead of the future one.

Definition 2.8 (Semantics of LTL). *Assume an atomic proposition set AP, a labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ and a state labeling function $L : \Sigma \to 2^{AP}$. The* semantics *of LTL is given by the satisfaction relation $\models_{LTL} \subseteq (\Sigma \cup \Pi) \times Form_{LTL}$ (or short $\models$) which evaluates LTL formulae in the context of a path as follows:* $\models_{LTL}$

$$
\begin{array}{lll}
\pi \models_{LTL} a & \textit{iff} & a \in L(\pi(0)) \\
\pi \models_{LTL} \varphi_1 \wedge \varphi_2 & \textit{iff} & (\pi \models_{LTL} \varphi_1) \wedge (\pi \models_{LTL} \varphi_2) \\
\pi \models_{LTL} \neg\varphi & \textit{iff} & \pi \not\models_{LTL} \varphi \\
\pi \models_{LTL} \mathcal{X}\varphi & \textit{iff} & \pi^1 \models_{LTL} \varphi \\
\pi \models_{LTL} \varphi_1 \mathcal{U}\,\varphi_2 & \textit{iff} & \exists j \geq 0.\ (\pi^j \models_{LTL} \varphi_2) \wedge \forall 0 \leq i < j.\ (\pi^i \models_{LTL} \varphi_1)\ .
\end{array}
$$

For a state $\sigma \in \Sigma$ and an LTL formula φ we define $\sigma \models_{LTL} \varphi$ to hold iff $\pi \models_{LTL} \varphi$ for all paths $\pi \in \Pi(\sigma)$ of $\mathcal{LSTS}$ starting in σ, and $\mathcal{LSTS} \models_{LTL} \varphi$ iff $\sigma_0 \models_{LTL} \varphi$ for all $\sigma_0 \in Init$.

Example 2.8. *We give some example LTL formulae and some paths of the system from Example 2.7 satisfying them. Thereby we omit labelings irrelevant for the satisfaction.*



The initial state σ_1 of the system $\mathcal{LSTS}$ from Example 2.7 does not satisfy $\mathcal{F}b$ (written $\sigma_1 \not\models_{LTL} \mathcal{F}b$), since there is a path $\pi = \sigma_1 \to \sigma_1 \to \ldots$ on which b never holds. But it satisfies $\mathcal{F}a$, since the proposition a holds in the initial state.

Remark 2.2. There are two special subclasses of path properties: safety and liveness properties. Intuitively, a *safety* property states that something "bad" never happens. E.g., the safety property $\mathcal{G}a$ expresses that the proposition a holds all the time. Partial correctness of a program—whenever the program terminates its output is correct—is also a safety property. The violation of a safety property by a path π can be shown by looking at a finite prefix of π.

In contrast, *liveness* properties express that something "good" will eventually happen. E.g., $\mathcal{F}a$ is a liveness property meaning that a will happen after a finite number of steps. Termination is a typical liveness property. To show the violation of a liveness property we must consider infinite paths.

Note that not all path formulae starting with the "globally" operator are safety properties. E.g., reactiveness, stating that something (let's say a) holds over and over again, is a typical liveness property which can be formalized as $\mathcal{GF}a$.

By definition, the sets of safety and liveness properties are disjoint. However, their union does not cover all path properties. E.g., program correctness can be expressed only by the conjunction of a safety property (partial correctness) and a liveness property (termination).

Remark 2.3. Besides the above notation, for the temporal operators there is another commonly used alternative notation:

$\mathcal{U}$	for	$\mathcal{U}$
$\bigcirc$	for	$\mathcal{X}$
$\Diamond$	for	$\mathcal{F}$
$\Box$	for	$\mathcal{G}$

For example, the formula $\mathcal{GF}\varphi$ can also be written as $\Box\Diamond\varphi$.

Computation Tree Logic (CTL)

Whereas LTL argues about linear paths, CTL formulae specify properties of computation trees. We distinguish between state formulae and path formulae. Intuitively, state formulae describe properties of the states (nodes) in the computation tree, and path formulae describe properties of paths in the tree. On the one hand, a path formula can be converted into a state formula by putting an existential or a universal quantifier in front of it, denoting that the path formula holds for a path respectively for all paths starting in a given node of the computation tree. On the other hand, state formulae are used to generate path formulae using the temporal operators. This implies, that a CTL state formula contains quantifiers and temporal operators in an alternating manner.

Definition 2.9 (Syntax of CTL). *Assume a set AP of atomic propositions.* CTL state formulae *can be built according to the abstract grammar*

$$\psi \quad ::= \quad a \mid (\psi \wedge \psi) \mid (\neg\psi) \mid (E\varphi) \mid (A\varphi)$$

with $a \in AP$ and where φ is a CTL path formula.
CTL path formulae *are built according to the abstract grammar*

$$\varphi \quad ::= \quad \mathcal{X}\psi \mid \psi\,\mathcal{U}\,\psi$$

where ψ is a CTL state formula.
CTL formulae *are CTL state formulae building the set $Form_{CTL}^{AP}$ (or short $Form_{CTL}$).* $Form_{CTL}$

We omit parentheses when it causes no confusion. Similarly to LTL, we can introduce the "finally" and "globally" operators. For state formulae ψ we define $\mathcal{F}\psi = true\,\mathcal{U}\,\psi$ as path formulae. Note that the LTL definition $\mathcal{G}\psi = \neg(true\,\mathcal{U}\,\neg\psi)$ of "globally" cannot be directly adapted to CTL, since it is not accepted by the CTL syntax. Instead, we define

$$\begin{aligned} E\mathcal{G}\psi &\leftrightarrow \neg A true\,\mathcal{U}\,\neg\psi \\ A\mathcal{G}\psi &\leftrightarrow \neg E true\,\mathcal{U}\,\neg\psi\ . \end{aligned}$$

Definition 2.10 (Semantics of CTL). *Assume an atomic proposition set AP, a labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ and a state labeling function $L : \Sigma \to 2^{AP}$. The satisfaction relation $\models_{CTL} \subseteq (\Sigma \cup \Pi) \times Form_{CTL}$ (or short $\models$) evaluates CTL state formulae in the context of a state, and CTL path formulae in the context of a path as follows:* $\models_{CTL}$

$$\begin{array}{lll} \sigma \models_{CTL} a & \textit{iff} & a \in L(\sigma) \\ \sigma \models_{CTL} \psi_1 \wedge \psi_2 & \textit{iff} & (\sigma \models_{CTL} \psi_1) \wedge (\sigma \models_{CTL} \psi_2) \\ \sigma \models_{CTL} \neg\psi & \textit{iff} & \sigma \not\models_{CTL} \psi \\ \sigma \models_{CTL} E\varphi & \textit{iff} & \exists \pi \in \Pi(\sigma).\ \pi \models_{CTL} \varphi \\ \sigma \models_{CTL} A\varphi & \textit{iff} & \forall \pi \in \Pi(\sigma).\ \pi \models_{CTL} \varphi \\ \\ \pi \models_{CTL} \mathcal{X}\psi & \textit{iff} & \pi(1) \models_{CTL} \psi \\ \pi \models_{CTL} \psi_1\,\mathcal{U}\,\psi_2 & \textit{iff} & \exists 0 \leq j.\ (\pi(j) \models_{CTL} \psi_2) \wedge \forall 0 \leq i < j.\ (\pi(i) \models_{CTL} \psi_1)\ . \end{array}$$

For $\psi \in Form_{CTL}$ we define $\mathcal{LSTS} \models_{CTL} \psi$ iff $\sigma_0 \models_{CTL} \psi$ for all $\sigma_0 \in Init$.

Example 2.9. *For our $\mathcal{LSTS}$ from Example 2.7 the CTL formula $A\mathcal{G}E\mathcal{X}b$ holds, since at each node of the computation tree we can take a transition to σ_2 labeled with b.*

The CTL formula $A\mathcal{G}E\mathcal{G}a$ does not hold, as the path $\sigma_1 \to \sigma_2 \to \sigma_2 \to \ldots$ violates the path property $\mathcal{G}E\mathcal{G}a$.

However, the CTL formula $A\mathcal{G}E\mathcal{X}E\mathcal{G}a$ holds.

Remark 2.4. Sometimes it is useful to define sub-logics of CTL that allow quantification in a restrictive manner: ACTL stays for the subset of CTL in which existential quantification cannot be expressed and ECTL for the one excluding universal quantification.

ACTL
ECTL

Remark 2.5. Additionally to the temporal operators in Remark 2.3, also quantifiers have an alternative notation:

$\exists$	$\exists$	for	E
$\forall$	$\forall$	for	A

For example, the formula $A\mathcal{G}E\mathcal{F}\varphi$ can also be written as $\forall\Box\exists\Diamond\varphi$.

CTL*

The logic CTL* is an extension of LTL and CTL and allows arbitrary alternation of path quantifiers and temporal operators.

Definition 2.11 (Syntax of CTL*). *Assume a set AP of atomic propositions.* CTL* state formulae *can be built according to the abstract grammar*

$$\psi \quad ::= \quad a \mid (\psi \wedge \psi) \mid (\neg\psi) \mid (E\varphi)$$

with $a \in AP$ and where φ is a CTL path formula.*
CTL* path formulae *are built according to the abstract grammar*

$$\varphi \quad ::= \quad \psi \mid (\varphi \wedge \varphi) \mid (\neg\varphi) \mid (\mathcal{X}\varphi) \mid (\varphi\,\mathcal{U}\,\varphi)$$

where ψ is a CTL state formula.*
$Form_{CTL^*}$ CTL* formulae *are CTL* state formulae building the set $Form^{AP}_{CTL^*}$ (or short $Form_{CTL^*}$).*

Again, we omit parentheses when it causes no confusion. As in CTL, we can define the "finally" and "globally" operators also for CTL* as syntactic sugar. Note that the universal quantification is not part of the CTL* syntax since $A\varphi$ can be defined as syntactic sugar by $\neg E\neg\varphi$.

Definition 2.12 (Semantics of CTL*). *Assume an atomic proposition set AP, a labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ and a state labeling function $L : \Sigma \to 2^{AP}$. CTL* state formulae are evaluated in the context of a state and CTL* path formulae in the context of a path by the satisfaction relation $\models_{CTL^*} \subseteq (\Sigma \cup \Pi) \times Form_{CTL^*}$ (or short $\models$) as follows:*

$\models_{CTL^*}$

$$\begin{array}{lll}
\sigma \models_{CTL^*} a & \textit{iff} & a \in L(\sigma) \\
\sigma \models_{CTL^*} \psi_1 \wedge \psi_2 & \textit{iff} & (\sigma \models_{CTL^*} \psi_1) \wedge (\sigma \models_{CTL^*} \psi_2) \\
\sigma \models_{CTL^*} \neg\psi & \textit{iff} & \sigma \not\models_{CTL^*} \psi \\
\sigma \models_{CTL^*} E\varphi & \textit{iff} & \exists \pi \in \Pi(\sigma).\ \pi \models_{CTL^*} \varphi
\end{array}$$

$$\begin{array}{lll}
\pi \models_{CTL^*} \psi & \textit{iff} & \pi(0) \models_{CTL^*} \psi \\
\pi \models_{CTL^*} \varphi_1 \wedge \varphi_2 & \textit{iff} & (\pi \models_{CTL^*} \varphi_1) \wedge (\pi \models_{CTL^*} \varphi_2) \\
\pi \models_{CTL^*} \neg\varphi & \textit{iff} & \pi \not\models_{CTL^*} \varphi \\
\pi \models_{CTL^*} \mathcal{X}\varphi & \textit{iff} & \pi^1 \models_{CTL^*} \varphi \\
\pi \models_{CTL^*} \varphi_1 \mathcal{U} \varphi_2 & \textit{iff} & \exists 0 \le j.\ (\pi^j \models_{CTL^*} \varphi_2) \wedge \forall 0 \le i < j.\ (\pi^i \models_{CTL^*} \varphi_1)\ .
\end{array}$$

For $\psi \in Form_{CTL^*}$ we define $\mathcal{LSTS} \models_{CTL^*} \psi$ iff $\sigma_0 \models_{CTL^*} \psi$ for all $\sigma_0 \in Init$.

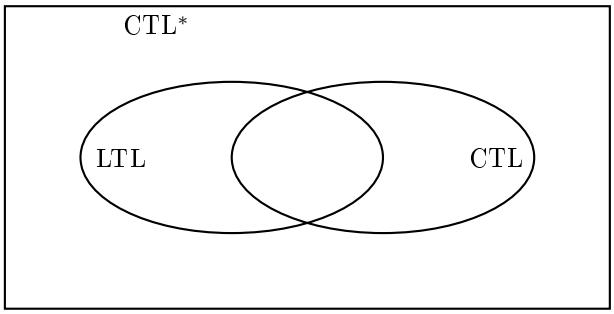


Figure 2.2: The expressiveness of LTL, CTL, and CTL*

The Relation of LTL, CTL, and CTL*

The logics LTL and CTL are incomparable, and both are included in CTL*, as shown in Figure 2.2. That LTL and CTL are incomparable means, that there are LTL formulae for which no equivalent CTL formulae exist, and vice versa, there are CTL formulae which are not expressible in LTL.

Example 2.10.

- *The LTL formula $\mathcal{FG}a$ is not expressible in CTL.*
- *The CTL formula $A\mathcal{FAG}a$ is not expressible in LTL.*

There are CTL* formulae that syntactically does not belong to LTL or to CTL but for that semantically equivalent LTL or CTL formulae can be given. However, CTL* is more expressive than LTL and CTL together, i.e., there are CTL* formulae that can be expressed neither in LTL nor in CTL (see Exercise **??**).

Example 2.11. *The CTL* formula $A\neg\mathcal{GE}\mathcal{F}a$ with $a \in AP$ is syntactically not a CTL formula. However, it can be expressed by the semantically equivalent CTL formula $A\mathcal{FAG}\neg a$.*

The CTL formula $A\mathcal{GA}\mathcal{F}\mathcal{G}a$ with $a \in AP$ is syntactically not an LTL formula. However, it can be expressed by the semantically equivalent LTL formula $\mathcal{FG}a$.*

2.3 CTL Model Checking for LSTSs

Model checking of discrete systems is not the basic content of this lecture, therefore here we restrict ourselves to the intuition behind explicit[1] CTL model checking for LSTSs, which can handle *finite-state* systems, only. This will be relevant later, as we will build *finite abstractions* of infinite-state systems to be able to apply model checking to them. For more details on model checking we refer to [BK08].

Given an LSTS, an atomic proposition set AP, a state labeling function and a CTL (state) formula ψ_0, CTL model checking labels the states of the LSTS recursively with the sub-state-formulae of ψ_0 inside-out, depending on the type of the subformula:

a: The labeling with atomic propositions $a \in AP$ is given by the labeling function.

[1] Explicit model checking is based on the enumeration of states, in contrast to symbolic model checking using a symbolic state representation like, e.g., binary decision diagrams (BDDs).

$\psi_1 \wedge \psi_2$: Given the labelings for ψ_1 and ψ_2, we label those states with $\psi_1 \wedge \psi_2$ that are labeled with both ψ_1 and ψ_2.

$\neg\psi$: Given the labeling for ψ, we label those states with $\neg\psi$ that are not labeled with ψ.

$E\mathcal{X}\psi$: Given the labeling for ψ, we label those states with $E\mathcal{X}\psi$ that have a successor state labeled with ψ.

$E\psi_1 \mathcal{U} \psi_2$: Given the labeling for ψ_1 and ψ_2, we

- label all with ψ_2 labeled states additionally with $E\psi_1 \mathcal{U} \psi_2$, and
- label those states that have the label ψ_1 and have a successor state with the label $E\psi_1 \mathcal{U} \psi_2$ also with $E\psi_1 \mathcal{U} \psi_2$ iteratively until a fixed point is reached, i.e., until no new labels can be added.

$A\mathcal{X}\psi$: Given the labeling for ψ, we label those states with $A\mathcal{X}\psi$ whose successor states are all labeled with ψ.

$A\psi_1 \mathcal{U} \psi_2$: Given the labeling for ψ_1 and ψ_2, we

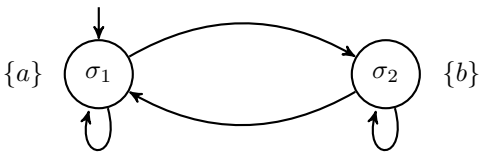
- label all with ψ_2 labeled states additionally with $A\psi_1 \mathcal{U} \psi_2$, and
- label those states that have the label ψ_1 and all of their successor states have the label $A\psi_1 \mathcal{U} \psi_2$ also with $A\psi_1 \mathcal{U} \psi_2$ iteratively until a fixed point is reached.

The formula ψ_0 is satisfied by the LSTS iff after termination of the procedure the initial state is labeled with ψ_0.

Since ψ_0 has only a finite number of sub-formulae and since there is only a finite number of states that can be labeled in the iterative cases, the procedure always terminates. Note that this model checking approach would not be complete, i.e., it would not terminate, for infinite-state systems.

Theorem 2.1 (Time complexity of CTL model checking for LSTS [BK08]). *Assume an LSTS $\mathcal{LSTS}$ with N states and K edges, an atomic proposition set AP, a state labeling function and a CTL formula ψ with M subformulae. Then the problem to decide whether $\mathcal{LSTS} \models_{CTL} \psi$ holds can be answered in time $\mathcal{O}((N+K) \cdot M)$.*

Example 2.12. *Assume again the LSTS from Example 2.7:*



In Example 2.9 we stated that this LSTS satisfies the CTL formula $A\mathcal{G}E\mathcal{X}E\mathcal{G}a$. Now we can prove this fact using model checking.

First we replace the syntactic sugar of the "globally" operator by its definition using

$$\begin{aligned} E\mathcal{G}\psi &\leftrightarrow \neg A true \mathcal{U} \neg\psi \\ A\mathcal{G}\psi &\leftrightarrow \neg E true \mathcal{U} \neg\psi. \end{aligned}$$

This yields

$$\psi := \neg\left(E true \mathcal{U} \neg\left(E\mathcal{X}\neg\left(A true \mathcal{U}\ (\neg a)\right)\right)\right)\ .$$

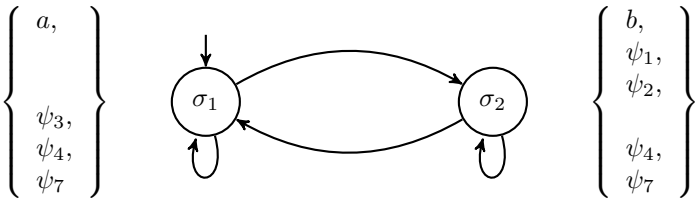
Model checking this property for the given system consists of labeling the states with the following subformulae in this order:

1. $\psi_1 := \neg a$
2. $\psi_2 := A true \mathcal{U} \psi_1$
3. $\psi_3 := \neg \psi_2$
4. $\psi_4 := E\mathcal{X}\psi_3$
5. $\psi_5 := \neg \psi_4$
6. $\psi_6 := E true \mathcal{U} \psi_5$
7. $\psi_7 := \neg \psi_6$

Labeling with the atomic proposition a is given by the labeling function: it holds only in σ_1. For the labeling with the above subformulae we get:

1. $\psi_1 := \neg a$*: We label with ψ_1 all those states where a does not hold. That means we label σ_2 with ψ_1.*
2. $\psi_2 := A true \mathcal{U} \psi_1$*:*
   - *We first label with ψ_2 all those states where ψ_1 holds. That means, we label σ_2 with ψ_2.*
   - *Those states that are not yet labeled with ψ_2 but whose successors are all labeled with ψ_2 get also labeled with ψ_2. However, there are no such states.*
3. $\psi_3 = \neg \psi_2$*: We label with ψ_3 all states that are not labeled with ψ_2. That means, we label σ_1 with ψ_3.*
4. $\psi_4 := E\mathcal{X}\psi_3$*: We label with ψ_4 all states that have a successor state labeled with ψ_3. That means, we label both σ_1 and σ_2 with ψ_4.*
5. $\psi_5 := \neg \psi_4$*: Label with ψ_5 all states that are not labeled with ψ_4. As both states are labeled with ψ_4, no states get the label ψ_5 attached.*
6. $\psi_6 := E true \mathcal{U} \psi_5$*:*
   - *We label with ψ_6 all states with the label ψ_5. However, there are no such states.*
   - *We label with ψ_6 all states that are not yet labeled with ψ_6 but that have a successor state labeled with ψ_6. There are no such states.*
7. $\psi_7 := \neg \psi_6$*: We label with ψ_7 all states that are not labeled with ψ_6. That means, we label both states σ_1 and σ_2 with ψ_7.*

The labeling result is as follows:



As the initial state is labeled with ψ_7, the LSTS satisfies ψ_7.

2.4 Discrete-Time Systems

Though discrete systems have no continuous components in their model, the real-time behavior of the modeled systems may nonetheless be relevant. Assume a controller executing a program. The program itself can be modeled as a discrete system, however, it may be critical if the program executes too long and the control values arrive too late.

If we want to model time without having a hybrid model, we can use a *discrete-time model*: Time is modeled by discrete time steps, also called *ticks*. Each transition step lasts for exactly one tick. Thus the elapsed time between two actions is always a multiple of a tick.

In order to describe the time behavior of discrete-time systems, the temporal operators of LTL, CTL, and CTL* can be extended with time bounds. This way we can express not only that some events take place but also *when* they take place in time. However, this extension does not increase the expressive power of the logics, i.e., a formula in the extended logics can be represented with an equivalent formula without the discrete-time extension. This has the effect that we can use model checking for LTL, CTL and CTL* also for their discrete-time extensions.

Remember that only the temporal operators "next" $\mathcal{X}$ and "until" $\mathcal{U}$ are basic, the remaining ones like "finally" $\mathcal{F}$ and "globally" $\mathcal{G}$ are syntactic sugar.

We extend the "next" operator $\mathcal{X}$ with an upper index. The formula $\mathcal{X}^k\varphi$ with $k \in \mathbb{N}$ denotes that φ is true after k steps. This indexed "next" operator does not increase the expressiveness of the logic, as it is syntactic sugar. In LTL it is defined recursively by

$$\mathcal{X}^k\varphi \quad = \quad \begin{cases} \varphi & \textit{if } k = 0 \\ \mathcal{X}\mathcal{X}^{k-1}\varphi & \textit{else.} \end{cases}$$

Thus $\mathcal{X}^k\varphi = \underbrace{\mathcal{X}\ldots\mathcal{X}}_{k}\varphi$ in LTL.

In CTL the quantifiers and temporal operators are alternating. For CTL we define

$$E\mathcal{X}^k\psi \quad = \quad \begin{cases} \psi & \textit{if } k = 0 \\ E\mathcal{X}E\mathcal{X}^{k-1}\psi & \textit{else.} \end{cases}$$

Thus $E\mathcal{X}^k\psi = \underbrace{E\mathcal{X}\ldots E\mathcal{X}}_{k}\psi$. The definition in combination with the universal quantifier $A\mathcal{X}^k\psi$ is analogous.

The extension of the "until" $\mathcal{U}$ operator is similar, but here we allow intervals instead of fixed values for the time bounds. The formula $\varphi_1\ \mathcal{U}^{[k_1,k_2]}\ \varphi_2$ ($k_1, k_2 \in \mathbb{N}$, $k_1 \leq k_2$) states that there exists a $k \in \mathbb{N}$ with $k_1 \leq k \leq k_2$ such that φ_2 holds in k steps and φ_1 holds all the time before. We also allow right-open intervals with k_2 being ∞, such that we can still represent the original "until" operator by $\varphi_1\ \mathcal{U}^{[0,\infty)}\ \varphi_2 = \varphi_1\ \mathcal{U}\ \varphi_2$.
In LTL we define

$$\varphi_1\ \mathcal{U}^I\ \varphi_2 \quad = \quad \begin{cases} \varphi_1\ \mathcal{U}\ \varphi_2 & \textit{for } I = [0,\infty) \\ \varphi_2 & \textit{for } I = [0,0] \\ \varphi_1 \wedge \mathcal{X}(\varphi_1\ \mathcal{U}^{[k_1-1,k_2-1]}\ \varphi_2) & \textit{for } I = [k_1,k_2],\ k_1 > 0 \\ \varphi_2 \vee (\varphi_1 \wedge \mathcal{X}(\varphi_1\ \mathcal{U}^{[0,k_2-1]}\ \varphi_2)) & \textit{for } I = [k_1,k_2],\ k_1 = 0, k_2 > 0\ . \end{cases}$$

In CTL we define

$$E\psi_1\ \mathcal{U}^I\ \psi_2 \quad = \quad \begin{cases} E\psi_1\ \mathcal{U}\ \psi_2 & \textit{for } I = [0,\infty) \\ \psi_2 & \textit{for } I = [0,0] \\ \psi_1 \wedge E\mathcal{X}E(\psi_1\ \mathcal{U}^{[k_1-1,k_2-1]}\ \psi_2) & \textit{for } I = [k_1,k_2],\ k_1 > 0 \\ \psi_2 \vee (\psi_1 \wedge E\mathcal{X}E(\psi_1\ \mathcal{U}^{[0,k_2-1]}\ \psi_2)) & \textit{for } I = [k_1,k_2],\ k_1 = 0, k_2 > 0\ . \end{cases}$$

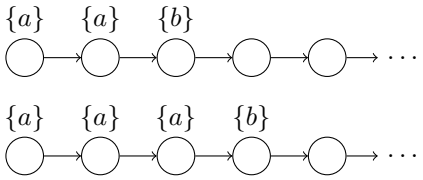
We also write

- $\mathcal{U}^{\leq k}$ instead of $\mathcal{U}^{[0,k]}$,
- $\mathcal{U}^{\geq k}$ for $\mathcal{U}^{[k,\infty]}$,
- $\mathcal{U}^{=k}$ for $\mathcal{U}^{[k,k]}$, and
- $\mathcal{U}$ for $\mathcal{U}^{[0,\infty]}$.

Example 2.13. *The discrete-time LTL formula $a\,\mathcal{U}^{[2,3]}\,b$ is defined as*

$$a \wedge \mathcal{X}(a \wedge \mathcal{X}(b \vee (a \wedge \mathcal{X}b))).$$

It is satisfied by paths of the following form:



As the discrete-time temporal operators are defined as syntactic sugar, model checking can be applied to check the validity of discrete-time temporal formulae for labeled state transition systems [Kat99, CGP01].

Chapter 3

General Hybrid Systems

3.1 Hybrid Systems

Discrete systems are systems with discrete, instantaneous state changes. E.g., when abstracting away physical details, a sensor reporting whether a tank is full or whether the temperature is above a certain threshold can be considered as a simple discrete system. Also a program running on a computer can be seen as a discrete system, when we assume that each atomic execution step changes the program's configuration in a discrete manner. Note that, though the state space of a program can be very large, due to the finite memory it is finite. Other systems might have an infinite or even uncountable state space, they are nevertheless classified as discrete systems when their *state changes* can be assumed to be discrete.

Dynamic systems are systems with a real-valued state space and *continuous* behavior. Physical systems with quantities like time, temperature, speed, acceleration etc. are dynamic systems. There evolution over time can be described by continuous functions or ordinary differential equations.

Hybrid systems are systems with combined discrete and continuous behavior (cf. Figure 3.1). Typical examples are physical systems controlled by a discrete controller. In modern cars there are hundreds of embedded digital chips helping to drive the car, that means, controlling the physical behavior like speed and acceleration. Behind the autopilot of an airplane there is a program running on a computer and acting with the physical environment.

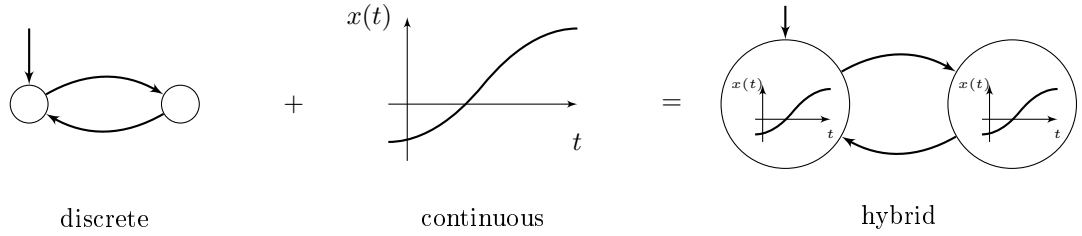


Figure 3.1: Hybrid systems exhibit a combined discrete-continuous behavior

In the following we introduce some hybrid system examples from [ACH+95, Hen96].

Example 3.1 (Thermostat). *Assume a thermostat, which senses the temperature x of a room and turns a heater on and off in order to keep the temperature between $17°C$ and $23°C$. Initially,*

the heater is on and the temperature is $20°C$. If the heater is on, the temperature increases according to the differential equation $\dot{x} = K(h - x)$ where $h \in \mathbb{R}_{>0}$ is a constant of the heater and $K \in \mathbb{R}_{>0}$ is a room constant. If the temperature is $22°C$ or above, but at latest when it reaches $23°C$, the heater gets turned off. If the heater is off, the temperature falls according to the differential equation $\dot{x} = -Kx$. If the temperature falls to $18°C$ or below, but at latest if it reaches $17°C$, the heater gets switched on. Figure 3.2 visualizes a possible behavior of the system, both of its continuous dynamics (the temperature) and its discrete control (the heater being on or off).

This system is hybrid. The discrete part of the system's state consists of the control mode of the heater being on or off. The continuous part is the temperature which continuously evolves over time, taking values from $\mathbb{R}$. The discrete part controls the continuous part by changing the discrete state and thereby influencing the continuous behavior.

Note that, since the heater gets switched on and off within certain temperature intervals, the system is non-deterministic. Replacing these intervals by fixed values would yield a deterministic system.

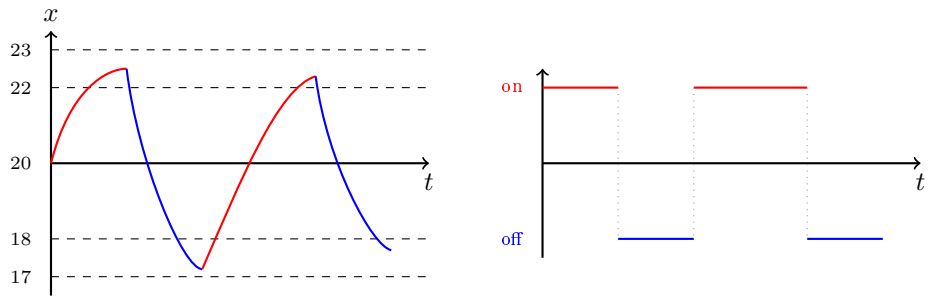


Figure 3.2: A possible behavior of the thermostat: the continuous dynamics for the temperature (left) and the control state with the heater being on or off (right) as a function of time

Example 3.2 (Water-level monitor). *Assume two identical, constantly leaking water tanks and a hose that refills exactly one of the tanks at each point in time (Figure 3.3 left). Let us denote the water level in the two tanks by x_1 and x_2, respectively, and let the leaking lead to a decrease of v_1 and v_2 units of tank height per time unit, respectively, for some $v_1, v_2 \in \mathbb{R}_{>0}$ without refilling. The hose fills $w \in \mathbb{R}_{>0}$ units of tank height per time unit. Thus the derivative of the water height for the first tank is $\dot{x}_1 = w - v_1$ when it gets refilled and $\dot{x}_1 = -v_1$ otherwise. The water height in the second tank changes according to $\dot{x}_2 = w - v_2$ when it gets refilled and $\dot{x}_2 = -v_2$ otherwise. When refilling the first tank, the hose switches to the second tank when its water level x_2 reaches a given lower threshold $r_2 \in \mathbb{R}_{>0}$. The switch from the second tank to the first one works analogously when x_1 reaches some $r_1 \in \mathbb{R}_{>0}$.*

Also this is a hybrid system. The discrete part of the state space consists of the position of the hose refilling either the first or the second tank. The continuous part of the state space corresponds to the water heights in the tanks which evolve continuously over time.

Example 3.3 (Bouncing ball). *Assume a bouncing ball with the initial height $h \in \mathbb{R}_{\geq 0}$ and with an initial upwards directed speed $v \in \mathbb{R}_{>0}$. Due to gravity, the ball has the acceleration $\dot{v} = -g$. Thus the ball's speed is decreasing to 0 until the ball reaches its highest position, and gets negative when the ball is falling down again. The ball bounces when it reaches the earth at*

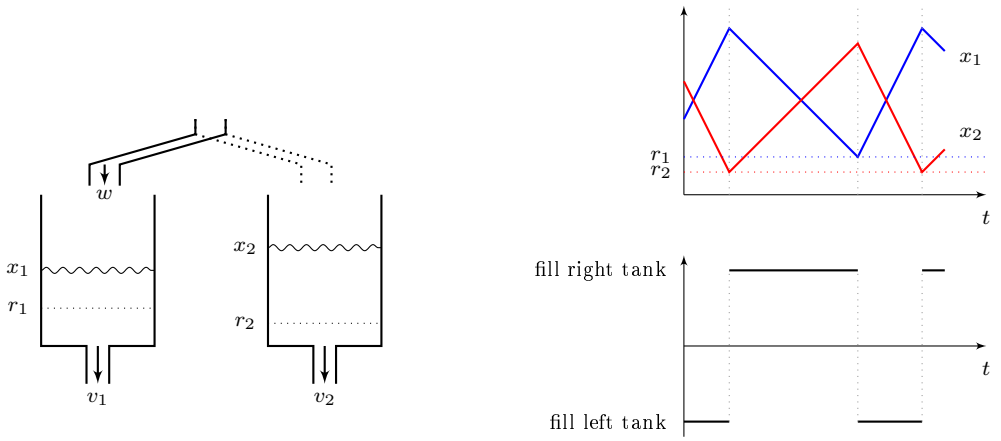


Figure 3.3: Water-level monitor illustration (left) and a possible behavior (right)

position $h = 0$ with a speed $v < 0$. When bouncing, the sign of v gets inverted, and a part of the ball's kinetic energy gets lost. Its speed after bouncing is $-cv$ with some $c \in (0, 1) \subseteq \mathbb{R}$ and v the speed before bouncing. Figure 3.4 illustrates the behavior of the system.

The continuous part of the state space covers the physical quantities of height and speed which follow the same evolution rules all the time. Thus there is only a single mode ("moving") for the ball behavior, and the state space does not have any discrete component. However, the discrete time points of bouncing introduce discrete events. That's why a bouncing ball can also be considered as a hybrid system.

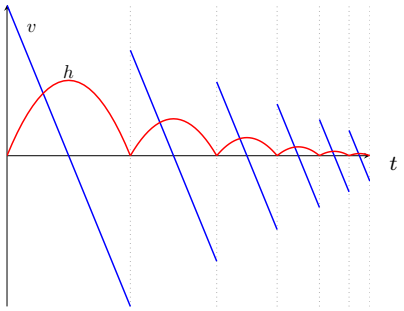


Figure 3.4: A possible behavior of the bouncing ball

3.2 Hybrid Automata

In an LTS the values of the variables may change instantaneously by taking a *discrete* transition from one location to another. Hybrid automata extend LTSs: Additionally to such discrete state changes, while control stays in a location, times passes by, and the values of variables change *continuously* according to some continuous functions. The combination of the discrete and the continuous behaviour leads to the term "hybrid".

$\mathcal{H}$ **Definition 3.1 (Syntax of hybrid automata).** *A hybrid automaton $\mathcal{H}$ is a tuple $(Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ where*

- *$(Loc, Var, Con, Lab, Edge, Init)$ is an LTS with real-valued variables Var, V the set of all valuations $\nu : Var \to \mathbb{R}$, and $\Sigma = Loc \times V$ the set of states,*
- *Act* *Act is a function assigning a set of activities $f : \mathbb{R}_{\geq 0} \to V$ to each location which are time-invariant meaning that $f \in Act(l)$ implies $(f+t) \in Act(l)$ where $(f+t)(t') = f(t+t')$ for all $t' \in \mathbb{R}_{\geq 0}$, and*
- *Inv* *a function Inv assigning an invariant $Inv(l) \subseteq V$ to each location $l \in Loc$.*

Compared to LTS, we have two new components: the activities and the invariants attached to the locations. The activities describe the continuous state changes in the locations when time passes by. The invariants restrict this behaviour such that time can evolve only as long as the invariant of the current location is satisfied. The control must leave the location before the invariant gets violated using a discrete transition. Also entering a location by a discrete step is only possible if the target location's invariant is satisfied after the step.

The execution of a hybrid automaton starts in a state $\sigma_0 = (\ell_0, \nu_0) \in Init$ from the initial set. The invariant $Inv(\ell_0)$ of the initial location ℓ_0 must be satisfied by the initial valuation ν_0, i.e., $\nu_0 \in Inv(\ell_0)$ must hold. Now two things can happen:

1. Time can pass by in the current location ℓ_0, and the values of the variables evolve according to a function $f : \mathbb{R}_{\geq 0} \to V$ from $Act(\ell_0)$. The function f must satisfy $f(0) = \nu_0$, i.e., it assigns the initial valuation to the time point 0. After t time units the variables' values are given by $\nu_1 = f(t)$, i.e., the system reaches the state (ℓ_0, ν_1).

   However, the control may stay in ℓ_0 only as long as the invariant $Inv(\ell_0)$ of ℓ_0 is satisfied. I.e., t time can pass by only if $\forall 0 \leq t' \leq t$ we have $f(t) \in Inv(\ell_0)$.

2. A discrete state change can happen if there is an enabled edge from ℓ_0, i.e., if there is a $(\ell_0, a, \mu, \ell_1) \in Edge$ and a valuation $\nu_1 \in V$ such that $(\nu_0, \nu_1) \in \mu$. The invariant of the target location must be satisfied after the step, i.e., $\nu_1 \in Inv(\ell_1)$ must hold.

From the state resulting from such a time or discrete step the system can again take either a time or a discrete step as described above.

Definition 3.2 (Semantics of hybrid automata). *The semantics of a hybrid automaton $\mathcal{H} = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ is given by an operational semantics consisting of two rules, one for the discrete instantaneous steps and one for the continuous time steps.*

1. *Discrete step semantics*

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

2. *Time step semantics*

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

An execution step

$$\rightarrow \;=\; \xrightarrow{a} \cup \xrightarrow{t}$$

of $\mathcal{H}$ is either a discrete or a time step. A path *(or run or excecution) π of $\mathcal{H}$ is a sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \ldots$ with $\nu_0 \in Inv(\ell_0)$ and $\sigma_i \rightarrow \sigma_{i+1}$ for all $i \geq 0$. We use $\Pi_{\mathcal{H}}(\sigma)$ (or short $\Pi(\sigma)$) to denote the set of all paths of $\mathcal{H}$ starting in σ. A state σ of $\mathcal{H}$ is* reachable *iff there is a run of $\mathcal{H}$ starting in an initial state of $\mathcal{H}$ and leading to σ.* π

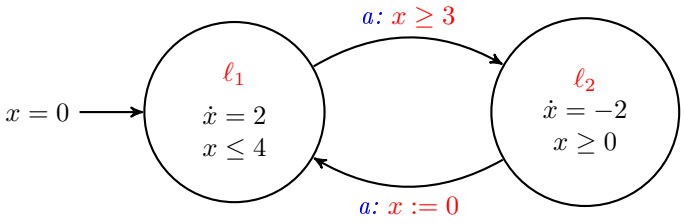
As it is the case for LTS, the operational semantics of hybrid automata define their induced state transition system. In the hybrid setting the set of reachables states is in general uncountable, as time progress leads to continuous behaviour.

As for LTS, guards and resets are often described logically by formulas respectively Boolean combinations of assignments (see page 14). Similarly to guards, also invariants can be specified by logical formulas over the variables.

Furthermore, the activities of a hybrid automaton are often given implicitly by differential equations, the activities being their solutions. E.g., $\dot{x} = 1$ specifies a set of activities $f : \mathbb{R}_{\geq 0} \rightarrow V$ with $f(t)(x) = t + c$ for some $c \in \mathbb{R}$ being the value of x at time point 0.

Finally, similarly to LTSs, also hybrid automata are often given in a graphical representation. We illustate the modeling by hybrid automata on our previous examples of the bouncing ball, the thermostat, and the water-level monitor. In the graphical representations in the following we omit the τ-transitions, non-synchronizing labels, trivial invariants, etc..

Example 3.4. *Assume the following graphical visualization of a hybrid automaton:*



The formal definition is as follows:

- $Loc = \{\ell_1, \ell_2\}$,
- $Var = \{x\}$,
- $Con(\ell_1) = Con(\ell_2) = \{x\}$,
- $Lab = \{\tau, a\}$,
- $Edge =$

$$\begin{aligned}\{\quad & (\ell_1,\ a,\ \{(\nu,\nu') \in V^2 \mid \nu(x) \geq 3 \wedge \nu'(x) = \nu(x)\},\ \ell_2),\\ & (\ell_2,\ a,\ \{(\nu,\nu') \in V^2 \mid \nu'(x) = 0\},\ \ell_1),\\ & (\ell_1,\ \tau,\ \{(\nu,\nu') \in V^2 \mid \nu = \nu'\},\ \ell_1),\\ & (\ell_2,\ \tau,\ \{(\nu,\nu') \in V^2 \mid \nu = \nu'\},\ \ell_2)\ \},\end{aligned}$$

- $Act(\ell_1) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = 2t + c\}$,
  $Act(\ell_2) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = -2t + c\}$,
- $Inv(\ell_1) = \{\nu \in V \mid \nu(x) \leq 4\}$,
  $Inv(\ell_2) = \{\nu \in V \mid \nu(x) \geq 0\}$,
- $Init = \{(\ell_1, \nu) \in \Sigma \mid \nu(x) = 0\}$.

Note that the activity sets for both locations are time-invariant. The instances of the discrete rule of the semantics for the two non-τ discrete transitions are:

$$\frac{\nu(x) \geq 3 \quad \nu'(x) = \nu(x) \quad (\nu'(x) \geq 0)}{(\ell_1, \nu) \xrightarrow{a} (\ell_2, \nu')} \; Rule^{\ell_1 \to \ell_2}_{discrete}$$

$$\frac{\nu'(x) = 0 \quad (\nu'(x) \leq 4)}{(\ell_2, \nu) \xrightarrow{a} (\ell_1, \nu')} \; Rule^{\ell_2 \to \ell_1}_{discrete}$$

The antecedents in parenthesis are implied by the other antecedents and are thus not needed. Since the only variable x is in the control variable sets of both locations, the τ-transitions do not allow any state change:

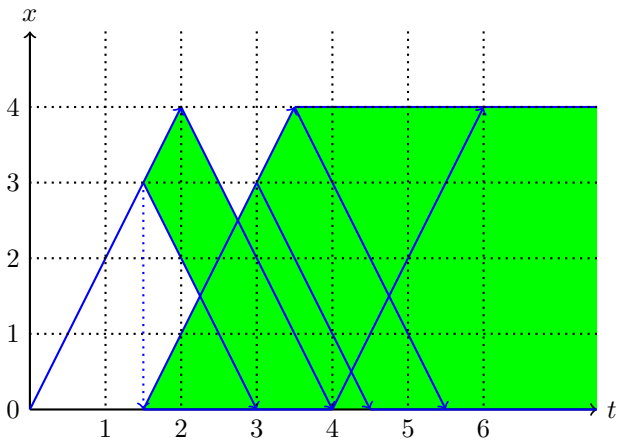
$$\frac{l \in Loc}{(l, \nu) \xrightarrow{\tau} (l, \nu)} \; Rule^{\tau}_{discrete}$$

For the time steps we have the following rule instances:

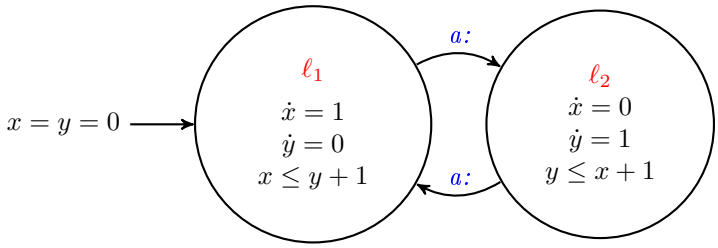
$$\frac{\nu'(x) \leq 4 \quad t \geq 0 \quad \nu'(x) = \nu(x) + 2t}{(\ell_1, \nu) \xrightarrow{a} (\ell_1, \nu')} \; Rule^{\ell_1}_{time}$$

$$\frac{\nu'(x) \geq 0 \quad t \geq 0 \quad \nu'(x) = \nu(x) - 2t}{(\ell_2, \nu) \xrightarrow{a} (\ell_2, \nu')} \; Rule^{\ell_2}_{time}$$

The following picture visualizes the behavior of the system by depicting the possible values for x at each point in time:



Example 3.5. *Assume another hybrid automaton:*



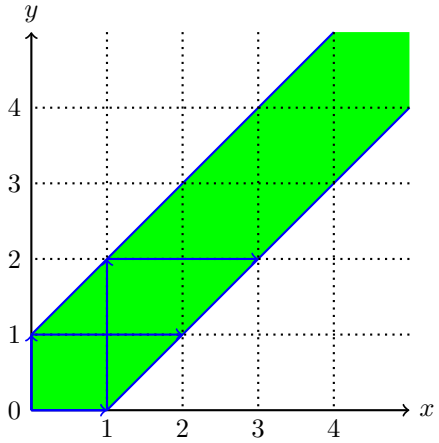
The formal definition is as follows:

- $Loc = \{\ell_1, \ell_2\}$,
- $Var = \{x, y\}$,
- $Con(\ell_1) = Con(\ell_2) = \{x, y\}$,
- $Lab = \{\tau, a\}$,
- $Edge =$

$$\begin{aligned}\{\quad & (\ell_1,\ a,\ \{(\nu,\nu') \in V^2 \mid \nu = \nu'\},\ \ell_2),\\ & (\ell_2,\ a,\ \{(\nu,\nu') \in V^2 \mid \nu = \nu'\},\ \ell_1),\\ & (\ell_1,\ \tau,\ \{(\nu,\nu') \in V^2 \mid \nu = \nu'\},\ \ell_1),\\ & (\ell_2,\ \tau,\ \{(\nu,\nu') \in V^2 \mid \nu = \nu'\},\ \ell_2)\ \},\end{aligned}$$

- $Act(\ell_1) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c_x, c_y \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = t + c_x \wedge f(t)(y) = c_y\}$,
  $Act(\ell_2) = \{f : \mathbb{R}_{\geq 0} \to V \mid \exists c_x, c_y \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = c_x \wedge f(t)(y) = t + c_y\}$,
- $Inv(\ell_1) = \{\nu \in V \mid \nu(x) \leq \nu(y) + 1\}$,
  $Inv(\ell_2) = \{\nu \in V \mid \nu(y) \leq \nu(x) + 1\}$,
- $Init = \{(\ell_1, \nu) \in \Sigma \mid \nu(x) = 0 \wedge \nu(y) = 0\}$.

The behaviour can be visualized as follows by depicting the reachable (x, y) value pairs (without representing the time):



Example 3.6 (Thermostat). *Assume again the thermostat from Example 3.1. The modeling hybrid automaton is depicted on Figure 3.5.*

In location ℓ_{on} the heater is on and the temperature raises according to the differential equation $\dot{x} = K(h - x)$. The location's invariant $x \leq 23$ assures that the heater turns off at latest when the temperature reaches $23°C$. Analogously for the location ℓ_{off}, where the heater is off.

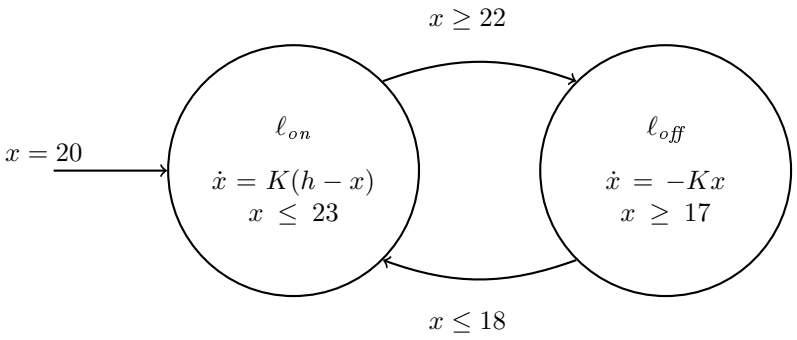


Figure 3.5: The hybrid automaton model of the thermostat

Control may move from location ℓ_{on} to ℓ_{off}, switching the heater off, if the temperature is at least $22°C$, and from ℓ_{off} to ℓ_{on} if the temperature is at most $18°C$. The temperature x does not change by jumping from ℓ_{on} to ℓ_{off} or from ℓ_{off} to ℓ_{on}. Initially, the heater is on and the temperature is $20°C$.

Note that this model is non-deterministic. E.g., in location ℓ_{on}, if the temperature is between $22°C$ and $23°C$, both time progress and switching the heater off are possible.

Example 3.7 (Water-level monitor). *The hybrid automaton model for the water-level monitor Example 3.2 is depicted in Figure 3.6.*

The automaton has two locations representing the control modes for refilling the first tank in ℓ_1 or refilling the second tank in ℓ_2. The water levels in the tanks are represented by the variables x_1 and x_2, being initially larger than r_1 resp. r_2 height units, i.e., initially $x_1 > r_1 \wedge x_2 > r_2$ holds.

Both tanks are leaking; the first tank looses v_1 height unit per time unit by leaking, the second tank v_2. When refilling a tank, w height unit per time unit is refilled. That means, the activities in ℓ_1 are represented by the differential equations $\dot{x}_1 = w - v_1$ and $\dot{x}_2 = -v_2$, and analogously for ℓ_2. In order to increase the water level when refilling a tank we assume $w > v_1$ and $w > v_2$.

The invariant $x_2 \geq r_2$ of ℓ_1 assures that the first tank is getting refilled only as long as there is enough water in the second tank (water level at least r_2). The hose will switch to refilling the second tank when the water level x_2 reaches r_2. This is done by taking the discrete transition from ℓ_1 to ℓ_2. Note that the transition's condition allows to switch only if x_2 is at most r_2, and the invariant assures that x_2 is at least r_2, such that the transition will be taken by the exact value r_2 of x_2. Refilling the second tank works analogously.

Note also that the discrete transitions can be taken only if the target location's invariant $x_1 \geq r_1$ is not violated. It can be shown that both invariants are globally valid, and thus the discrete transitions are never blocked by the invariants.

Example 3.8 (Bouncing Ball). *The hybrid automaton model of the bouncing ball from Example 3.3 is depicted on Figure 3.7. Initially the height of the ball x_1 is larger or equal 0 (height 0 corresponds to the earth and positive height above the earth) and its speed x_2 is positive, stating that the ball is initially raising.*

The automaton has a single location ℓ_0. Time progress in this location corresponds to the raising and falling of the ball. The differential equation $\dot{x}_1 = x_2$ defines x_2 as the derivative of the height, i.e., the ball's speed, and $\dot{x}_2 = -g$ with g the gravity constant defines the speed change

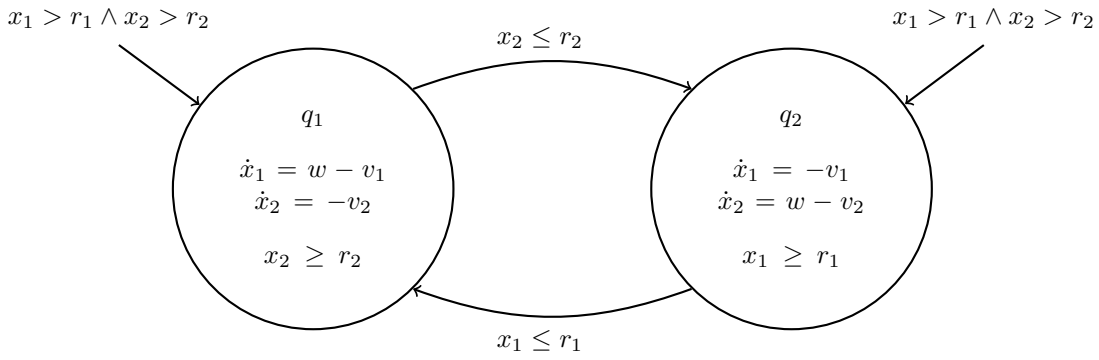


Figure 3.6: The hybrid automaton model of the water-level monitor

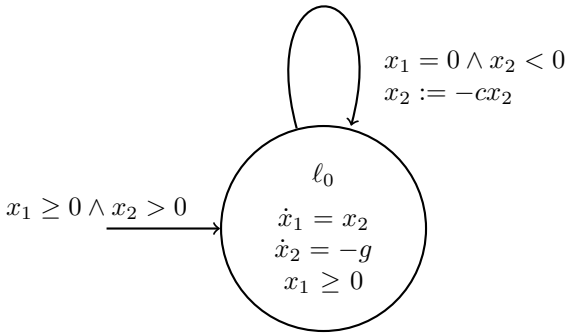


Figure 3.7: The hybrid automaton model of the bouncing ball

due to gravity.

The ball can raise and fall only as long as it has a non-negative height as stated by the invariant $x_1 \geq 0$*. After raising and reaching the highest point, it starts falling and reaches the earth when* $x_1 = 0$ *and* $x_2 < 0$*. Then it bounces, represented by the single discrete transition. Note that the bounce is forced by the invariant. The bounce changes the speed's direction and reduces its absolute value due to some loss of kinetic energy during bouncing as denoted by* $x_2 := -cx_2$*. After bouncing,* x_1 *is still* 0 *but* x_2 *is now positive, and the ball raises again.*

For the ease of modeling, also hybrid systems can be modeled componentwise. The resulting global system is given by the parallel composition of the different components. The parallel composition of hybrid automata extends the definition of the parallel composition for LTSs as follows.

Definition 3.3 (Parallel composition of hybrid automata). *Let*

$$\begin{aligned} \mathcal{H}_1 &= (Loc_1, Var, Con_1, Lab_1, Edge_1, Act_1, Inv_1, Init_1) \text{ and} \\ \mathcal{H}_2 &= (Loc_2, Var, Con_2, Lab_2, Edge_2, Act_2, Inv_2, Init_2) \end{aligned}$$

be two hybrid automata. The parallel composition *or* product $\mathcal{H}_1||\mathcal{H}_2$ *of* $\mathcal{H}_1$ *and* $\mathcal{H}_2$ *is defined* $\mathcal{H}_1||\mathcal{H}_2$

to be the hybrid automaton

$$\mathcal{H} \quad = \quad (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$$

with

- *The LTS part* $(Loc, Var, Con, Lab, Edge, Init)$ *equals the parallel composition*

  $$(Loc_1, Var, Con_1, Lab_1, Edge_1, Init_1)||(Loc_2, Var, Con_2, Lab_2, Edge_2, Init_2)$$

  of the LTS parts of the components,
- $Act(\ell_1, \ell_2) = Act_1(\ell_1) \cap Act_2(\ell_2)$ *for all* $(\ell_1, \ell_2) \in Loc$*, and*
- $Inv(\ell_1, \ell_2) = Inv_1(\ell_1) \cap Inv_2(\ell_2)$ *for all* $(\ell_1, \ell_2) \in Loc$*.*

In the following chapters we consider different subclasses of hybrid automata with increasing expressivity.

Chapter 4

Timed Automata

The popular modeling formalism of *timed automata* combines labeled transition system models with a notion of *time* as the only continuous component. Its success is based on two main facts: Firstly, this model class, despite its rather weak expressiveness, already allows to model a wide range of real-time systems. Secondly, the model checking problem for safety and liveness properties of timed automata is still efficiently decidable. Driven by both academic and industrial interests, a lot of effort was put into tool support. Uppaal is one of the most widely used tools for model checking timed automata.

In this chapter we first introduce *timed automata* in Section 4.1. In Section 4.2 we extend the logic CTL with continuous-time aspects, resulting in the logic *timed CTL* (*TCTL*). In this book we restrict ourselves to the introduction of TCTL. Another popular timed temporal logic is, e.g., metric LTL (MTL). We discuss *model checking* TCTL properties of timed automata in Section 4.3. For further reading on timed automata and its model checking algorithm we refer to [BK08].

4.1 Syntax and Semantics

A timed automaton has a finite number of *clocks* as variables. A clock measures the time, i.e., it continuously evolves at rate 1. The values of the clocks can only be accessed in a limited way. For read access, the only fact we can observe about a clock value is the result of a comparison of its value with a constant. Such comparisons can be formulated by *clock constraints*. For write access, clocks can only be *reset*, i.e., their values can only be set to 0.

Definition 4.1 (Syntax of clock constraints). Clock constraints *over a finite set $\mathcal{C}$ of clocks can be built using the following abstract grammar:*

$$g \quad ::= \quad x < c \mid x \leq c \mid x > c \mid x \geq c \mid g \wedge g$$

where $c \in \mathbb{N}$[1] *and $x \in \mathcal{C}$.*
Clock constraints which are not a conjunction are called atomic. *The set of atomic clock constraints over a set $\mathcal{C}$ of clocks is denoted by $ACC_{\mathcal{C}}$. The set of all clock constraints over $\mathcal{C}$ is referred to as $CC_{\mathcal{C}}$.*

Clock constraints are evaluated in the context of a *valuation* $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ assigning non-negative real values to clocks. We use $V_{\mathcal{C}}$ (or short V) for the set of all valuations.

[1] We could also allow $c \in \mathbb{Q}$.

Definition 4.2 (Semantics of clock constraints). *The semantics of clock constraints over a finite set $\mathcal{C}$ of clocks is given by the relation $\models_{cc} \subseteq V \times CC_{\mathcal{C}}$ (or short $\models$) defined as follows:*

$\models_{cc}$

$$\begin{array}{lll} \nu \models_{cc} x < c & \textit{iff} & \nu(x) < c \\ \nu \models_{cc} x \leq c & \textit{iff} & \nu(x) \leq c \\ \nu \models_{cc} x > c & \textit{iff} & \nu(x) > c \\ \nu \models_{cc} x \geq c & \textit{iff} & \nu(x) \geq c \\ \nu \models_{cc} g_1 \wedge g_2 & \textit{iff} & (\nu \models_{cc} g_1) \wedge (\nu \models_{cc} g_2) \ . \end{array}$$

For the sake of readability we also use notations like

$$\textit{true}, \quad x \in [c_1, c_2), \quad c_1 \leq x < c_2, \quad x = c, \ldots$$

with the expected meaning. E.g., $x = c$ can be defined using $x \geq c \wedge x \leq c$.

Based on its semantics, a clock constraint $g \in CC_{\mathcal{C}}$ can also be seen as the set $\{\nu \in V \mid \nu \models g\}$ of all valuations that satisfy g.

Remark 4.1. Note that the syntax of clock constraints allows conjunction but no negation, assuring that the sets defined by clock constraints are *convex*. This has the big advantage that, when we start time progress with a valuation $\nu \in V$ satisfying a clock constraint $g \in CC_{\mathcal{C}}$ then, since time progress is linear, when a valuation $\nu + t$ after some time elapse $t \in \mathbb{R}_{\geq 0}$ still satisfies the clock constraint g then we know that all the valuations inbetween also satisfied g, i.e., $\nu + t' \models g$ for all $0 \leq t' \leq t$.

As mentioned above, write access to clocks is restricted to resetting their values to 0.

Definition 4.3 (Syntax of clock reset). *Given a finite set $\mathcal{C}$ of clocks, a* clock reset *is an expression of the form $reset(C)$ with $C \subseteq \mathcal{C}$.*

$reset(C)$

Sometimes we also write $reset(x_1, \ldots, x_n)$ instead of $reset(\{x_1, \ldots, x_n\})$.

Also the semantics of a clock reset is given in the context of a valuation. Semantically, a clock reset $reset(C)$ denotes that the values of all clocks in C get reset to 0, and the values of all other clocks from $\mathcal{C} \backslash C$ remain unchanged.

Definition 4.4 (Semantics of clock reset). *Let $\mathcal{C}$ be a finite set of clocks and $C \subseteq \mathcal{C}$. The result of $reset(C)$ applied to a valuation $\nu \in V$ is given by the valuation satisfying*

$reset(C)$ *in* ν

$$(reset(C) \textit{ in } \nu)(x) = \begin{cases} 0 & \textit{if } x \in C \\ \nu(x) & \textit{otherwise} \end{cases}$$

for all $x \in \mathcal{C}$.

The following notation formalizes time delay.

$\nu + t$

Definition 4.5. *For all valuations $\nu \in V$ and constants $t \in \mathbb{R}_{\geq 0}$ we define the valuation $\nu + t$ by $(\nu + t)(x) = \nu(x) + t$ for all $x \in \mathcal{C}$.*

Example 4.1 (Clock access). *Assume a clock set $\mathcal{C} = \{x, y\}$ and a valuation $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ with $\nu(x) = 2$ and $\nu(y) = 3$. Then*

- *$\nu + 9$ assigns 11 to x and 12 to y,*
- *$reset(x)$ in $(\nu + 9)$ assigns 0 to x and 12 to y,*
- *$(reset(x)$ in $\nu) + 9$ assigns 9 to x and 12 to y,*
- *$reset(x)$ in $(reset(y)$ in $\nu)$ assigns 0 to both x and y, and*
- *$reset(x, y)$ in ν assigns 0 to both x and y.*

Next we give the definition of timed automata. These models have an LTS component with a rectricted syntax for discrete steps, allowing only clock constraints and clock resets in the definition of the transition relation. This discrete model is extended by introducing time as a continuous quantity: while the control stays in a location, time ellapses and the values of the clocks increase continuously. The main differences to LTS models are the following:

- The variable set of a timed automaton is denoted by $\mathcal{C}$ instead of *Var* to express the fact that all variables of a timed automaton are *clocks*. assign the value 0 to all clocks. A *state* of a timed automaton is a location-valuation pair $(l, \nu) \in Loc \times V_{\mathcal{C}} = \Sigma$, storing the location of the timed automaton in that the control currently stays together with the current values of the clocks.
- In order to restrict the transition relation of the discrete edges to the less powerful clock access, we use enabling conditions in form of clock constraints combined with reset sets in place of general transition relations. Given a pair $(g, C) \in CC_{\mathcal{C}} \times 2^{\mathcal{C}}$ of a clock constraint g and a reset set C, the corresponding *transition relation* $\mu \subseteq V^2$ is given by

$$\mu = \{(\nu, \nu') \in V^2 \mid \nu \models g \wedge \nu' = reset(C) \text{ in } \nu\} .$$

  Thus edges have the form $(\ell, a, (g, C), \ell') \in Loc \times Lab \times (CC_{\mathcal{C}} \times 2^{\mathcal{C}}) \times Loc)$.
- As long as the control stays in a location, the values of all clocks *evolve* with the derivative 1. That means, when a location is entered with a valuation ν, after t time the valuation will be $\nu + t$.
- The locations can be annotated with *invariants*. Control may stay in a location only as long as the invariant of the location is not violated. Invariants allow to enforce discrete transitions; without invariants, the control could stay in a location forever. Similarly to the guards of the discrete transitions, also invariants are defined by clock constraints.
- There is a further difference between LTS and timed automata regarding the parallel composition. For LTS the parallel composition supports shared variables accessible by different components. However, allowing shared variables in the timed automata composition would lead to some complications, which we do not discuss here. Instead, we restrict the composition of timed automata to components having disjoint variable sets. Note that when excluding shared variables, the only way of communication is label synchronization. Thus the definition of the controlled variable sets *Con* and also the τ-transitions get superfluous.

Definition 4.6 (Syntax of timed automata). *A* timed automaton *is a tuple* $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ *with*

- Loc *is a finite set of* locations,
- $\mathcal{C}$ *is a finite set of real-valued variables called* clocks,
- Lab *is a finite set of* synchronization labels,
- $Edge \subseteq Loc \times Lab \times (CC_{\mathcal{C}} \times 2^{\mathcal{C}}) \times Loc$ *is a finite set of* edges,
- $Inv : Loc \to CC_{\mathcal{C}}$ *is a function assigning an* invariant *to each location, and*
- $Init \subseteq Loc \times V_{\mathcal{C}} = \Sigma$ *a set of* initial states *with* $\nu(x) = 0$ *for all* $(\ell, \nu) \in Init$ *and each* $x \in \mathcal{C}$.

To simplify the formalisms, we extend the notations for valuations to states and use $\sigma \models_{CC} g$ for a state $\sigma = (\ell, \nu)$ to express that $\nu \models_{CC} g$. Similarly, for $\sigma = (\ell, \nu)$ we also write $reset(x)$ *in* σ to denote $(\ell, reset(x) \text{ in } \nu)$.

Definition 4.7 (Semantics of timed automata). *The* operational semantics *of a timed automaton $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ is given by the following two rules:*

$$\frac{(\ell, a, (g, C), \ell') \in Edge \quad \nu \models g \quad \nu' = reset(C) \text{ in } \nu \quad \nu' \models Inv(\ell')}{(\ell, \nu) \xrightarrow{a} (\ell', \nu')} \; Rule_{discrete}$$

$$\frac{t \in \mathbb{R}_{\geq 0} \quad \nu' = \nu + t \quad \nu' \models Inv(\ell)}{(\ell, \nu) \xrightarrow{t} (\ell, \nu')} \; Rule_{time} \; .$$

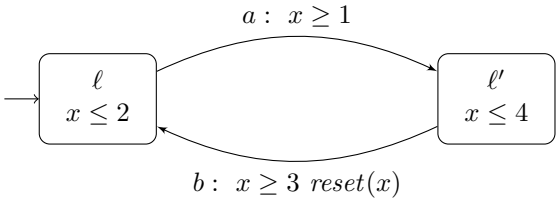
We write $\sigma \to \sigma'$ instead of $\sigma \xrightarrow{a} \sigma'$ or $\sigma \xrightarrow{t} \sigma'$ when the type of the step is not of interest.
A run *(or* path *or* execution*) of $\mathcal{T}$ is an infinite sequence $\sigma_0 \to \sigma_1 \to \sigma_2 \ldots$ with $\sigma_i \in \Sigma$ and $\sigma_0 = (\ell_0, \nu_0) \in Inv(\ell_0)$; if additionally $\sigma_0 \in Init$ then we call π an* initial path.
We use $\Pi_{\mathcal{T}}(\sigma)$ (or short $\Pi(\sigma)$) to denote the set of all paths of $\mathcal{T}$ starting in $\sigma \in \Sigma$, and define $\Pi_{\mathcal{T}} = \bigcup_{\sigma \in \Sigma} \Pi_{\mathcal{T}}(\sigma)$ (or short Π). A state is reachable *if there is an initial path leading to it; we write $Reach_{\mathcal{T}}$ (or short $Reach$) for the set of reachable states of $\mathcal{T}$.*

Note that, since the invariants are convex sets, it is enough to require that they hold *after* each time step, and we do not need the requirement that they hold *during* the whole period of a time step. Together with the requirement that the starting states of paths satisfy the corresponding invariants, we get by induction that the invariants hold on all paths at each time point.

Again, the semantics of a timed automaton induces an LSTS for its (in general uncountable) state space. As in the case of discrete systems, also timed automata can be augmented by a *labeling function*. However, since the state space is now uncountable, we attach propositions to the locations (instead of the states) by a labeling function $L : Loc \to 2^{AP}$ where AP denotes the set of atomic propositions. To simplify the notation, we overload the labeling function defining $L : \Sigma \to 2^{AP}$ with $L((\ell, \nu)) = L(\ell)$.

Timed automata are often represented graphically, where non-synchronizing labels, trivial conditions and empty reset sets are skipped. As all clocks evolve with derivative 1 we do not represent the time behavior in the graphs.

Example 4.2. *The graphical representation*



denotes the timed automaton $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ with

- $Loc = \{\ell, \ell'\}$
- $\mathcal{C} = \{x\}$,
- $Lab = \{a, b\}$,
- $Edge = \{(\ell, \; a, \; (x \geq 1, \; \emptyset), \; \ell'), \; (\ell', \; b, \; (x \geq 3, \; \{x\}), \; l)\}$,
- $Inv(\ell) = x \leq 2$, $Inv(\ell') = x \leq 4$,
- $Init = \{(\ell, \nu_0)\}$ *with* $\nu_0(x) = 0$.

Definition 4.8 (Parallel composition of timed automata). *Let $\mathcal{T}_1 = (Loc_1, \mathcal{C}_1, Lab_1, Edge_1, Inv_1, Init_1)$ and $\mathcal{T}_2 = (Loc_2, \mathcal{C}_2, Lab_2, Edge_2, Inv_2, Init_2)$ two timed automata with $\mathcal{C}_1 \cap \mathcal{C}_2 = \emptyset$.*

The parallel composition $\mathcal{T}_1||\mathcal{T}_2$ *is a timed automaton* $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ *with valuations* $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$*, valuation set* V*, and states* $\Sigma = Loc \times V$*, where*

- $Loc = Loc_1 \times Loc_2$
- $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$
- $Lab = Lab_1 \cup Lab_2$
- $Inv((\ell_1, \ell_2)) = Inv_1(\ell_1) \wedge Inv_2(\ell_2)$ *for all* $(\ell_1, \ell_2) \in Loc$
- $Init = \{((\ell_1, \ell_2), \nu) \in \Sigma \mid (\ell_1, \nu) \in Init_1 \wedge (\ell_2, \nu) \in Init_2\}$
- $Edge =$

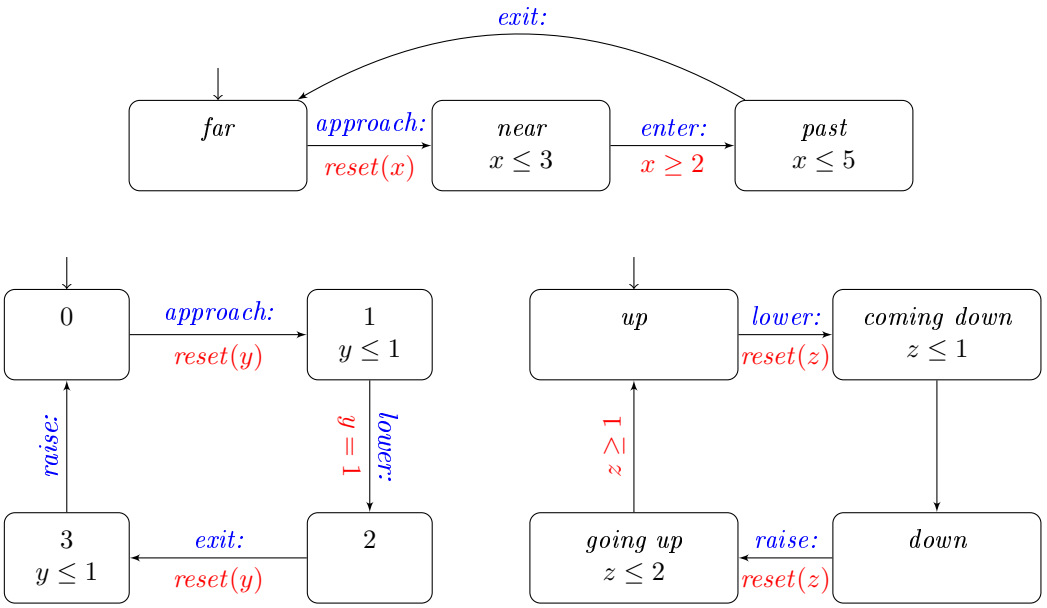
$$
\begin{array}{rcl}
\{((\ell_1,\ell_2), a, (g_1 \wedge g_2, \mathcal{C}_1 \cup \mathcal{C}_2), (\ell_1', \ell_2')) & \mid & (\ell_1, a, (g_1, \mathcal{C}_1), \ell_1') \in Edge_1 \wedge \\
 & & (\ell_2, a, (g_2, \mathcal{C}_2), \ell_2') \in Edge_2\} \\
\{((\ell_1,\ell_2), a, (g, \mathcal{C}), (\ell_1', \ell_2)) & \mid & (\ell_1, a, (g, \mathcal{C}), \ell_1') \in Edge_1 \wedge a \notin Lab_2\} \\
\{((\ell_1,\ell_2), a, (g, \mathcal{C}), (\ell_1, \ell_2')) & \mid & (\ell_2, a, (g, \mathcal{C}), \ell_2') \in Edge_2 \wedge a \notin Lab_1\} \ .
\end{array}
$$

To illustrate the parallel composition, we extend our previous LSTS railroad crossing model to a timed automaton model.

Example 4.3 (Railroad crossing). *We extend Example 2.3 with real-time behavior as follows:*

- *After the train triggers the "approach" signal it reaches the gate between* 2 *and* 3 *minutes. It passes the track between the "approach" and the "exit" sensors within* 5 *minutes. The timed automaton* $\mathcal{H}_{Train}$ *modeling the train has a clock* x *which is a control variable in each location.*
- *After receiving an "approach" signal, the controller delays* 1 *minute before it sends a "lower" signal to the gate. After receiving an "exit" signal it notifies the gate by emitting a "raise" signal with a delay of at most one minute. The timed automaton model* $\mathcal{H}_{Controller}$ *of the controller has a clock* y *being a control variable in each location.*
- *The gate needs at most one minute to be lowered and between one and two minutes to be raised. The timed automaton model* $\mathcal{H}_{Gate}$ *has its own clock* z*, being a control variable in each location.*

Adapting the syntax of timed automata we get the following graphical representation:



4.1.1 Continuous-Time Phenomena

Similarly to LTS, the semantics of a timed automaton induces a LSTS. Each path in the induced LSTS corresponds to a possible system behavior. However, some of the paths may model unrealistic behavior.

Time convergence: There are syntactically inavoidable paths of timed automata along which time converges, i.e., time never evolves beyond some value. For example, the timed automaton from Example 4.2 has a path

$$(\ell, \nu_1) \xrightarrow{1} (\ell, \nu_2) \xrightarrow{1/2} (\ell, \nu_3) \xrightarrow{1/4} (\ell, \nu_4) \xrightarrow{1/8} \dots$$

starting in the initial state and executing time steps with durations converging to 0. The time duration $\sum_{i=1}^{n} \frac{1}{i}$ converges to 2 with path length $n \to \infty$. Such a path is called *time-convergent*. Paths that are not time-convergent are called *time-divergent*.

Time-convergent paths are not realizable, but are unavoidable in the modeling. We will explicitly exclude such paths in the semantics of the logic TCTL for the property specification.

Timelock: There could be states in the LSTS of a timed automaton from which all paths are time-convergent, such that there is no possibility that time progresses forever. Such states do not allow time divergence, and are therefore called *timelock* states. Timed automata in which no timelock states are reachable are called *timelock-free*. Timelocks are modeling flaws, i.e., they can be avoided by appropriate modeling.

Zeno paths: Paths along which infinitely many discrete steps are performed in a finite amount of time are called *Zeno* paths. Note that all Zeno paths are time-convergent. Zeno paths are not realizable, as they would require infinitely fast processors. They are also modeling flaws and can be avoided by careful modeling.

Next we formalize the above properties.

Definition 4.9 (Time convergence, timelock, Zeno paths). *For a timed automaton* $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ *we define the time duration of a step by the function ExecTime* : $(Lab \cup \mathbb{R}_{\geq 0}) \to \mathbb{R}_{\geq 0}$ *with* *ExecTime*

$$ExecTime(\alpha) \quad = \quad \begin{cases} 0 & \text{if } \alpha \in Lab \\ \alpha & \text{if } \alpha \in \mathbb{R}_{\geq 0} \end{cases} .$$

The time duration of an infinite path $\pi = \sigma_0 \xrightarrow{\tau_0} \sigma_1 \xrightarrow{\tau_1} \sigma_2 \xrightarrow{\tau_2} \ldots$ *of* $\mathcal{T}$ *is defined by the (overloaded) function*

$$ExecTime(\pi) \quad = \quad \sum_{i=0}^{\infty} ExecTime(\tau_i).$$

- *An infinite path* $\pi \in \Pi$ *is said to be* time-divergent *if* $ExecTime(\pi) = \infty$*, and* time-convergent *otherwise.*
  For a state $\sigma \in \Sigma$ *we define* $\Pi_{div}(\sigma) \subseteq \Pi(\sigma)$ *to be the set of time-divergent infinite paths starting in* σ*, and* $\Pi_{div} = \bigcup_{\sigma \in \Sigma} \Pi_{div}(\sigma)$. $\Pi_{div}(\sigma), \Pi_{div}$
- *A state* $\sigma \in \Sigma$ *contains a* timelock *iff* $\Pi_{div}(\sigma) = \emptyset$*. A timed automaton is said to be* timelock-free *if none of its reachable states contains a timelock.*
- *An infinite path* $\pi \in \Pi$ *is said to be* Zeno *if it is time-convergent and infinitely many discrete actions are executed within* π*. The timed automaton* $\mathcal{T}$ *is said to be* non-Zeno *if it has no Zeno paths.*

As mentioned above, Zeno paths a modeling flows. To check whether a timed automaton is non-Zeno is algorithmically difficult. However, there is a sufficient (but not necessary) condition which is simple to check.

Theorem 4.1 (Sufficient condition for non-Zenoness). *Assume a timed automaton* $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ *such that for each sequence of edges*

$$\ell_0 \xrightarrow{\alpha_1 : g_1, C_1} \ell_1 \xrightarrow{\alpha_2 : g_2, C_2} \ell_2 \ldots \xrightarrow{\alpha_n : g_n, C_n} \ell_n = \ell_0$$

in $\mathcal{T}$ *there exists a clock* $x \in \mathcal{C}$ *such that*

1. $x \in C_i$ *for some* $0 < i \leq n$ *and*
2. *for all valuations* $\nu \in V$ *there exists a* $c \in \mathbb{N}_{>0}$ *such that*

$$\nu(x) < c \quad \rightarrow \quad (\nu \not\models g_j \text{ or } \nu \not\models Inv(\ell_j))$$

for some $0 < j \leq n$.

Then $\mathcal{T}$ *is non-Zeno.*

4.2 Timed Computation Tree Logic (TCTL)

Timed automata often model real-time systems that are *time-critical* in the sense that for their correct functioning certain events must occur within some time bounds. For example, in case of an accident the airbag of a car must react within very tight time limits. Also other controller are supposed to support control values within some predefined time bounds.

The untimed logics of the previous section are not yet able to argue about such time constraints. In this section we extend them for this purpose. Thereby we restrict ourselves to the extension of CTL to *timed CTL* (*TCTL*). The extensions of LTL and CTL* are analogous.

The main differences between CTL and TCTL are as follows:

- For discrete systems we used an atomic proposition set and a labeling function to assign atomic propositions to states. Besides such atomic propositions, for timed automata we also want to argue about clock values in form of atomic clock constraints. Therefore, both atomic propositions and atomic clock constraints are atomic TCTL state formulae.
- Since timed automata model continuous time, there is no "next" operator in TCTL.
- Remember that a CTL "until" formula $\psi_1 \mathcal{U} \psi_2$ is satisfied by a path if ψ_2 is satisfied by a state somewhere on the path, and ψ_1 holds in all the states before. In TCTL, the "until" operator of CTL gets indexed with a time interval. TCTL "bounded until" formulae have the form $\psi_1 \mathcal{U}^{[t_1,t_2]} \psi_2$, where the time interval $[t_1, t_2]$ puts a restriction *when* ψ_2 gets valid. A path satisfies the formula $\psi_1 \mathcal{U}^{[t_1,t_2]} \psi_2$ if, when measuring the time from the beginning of the path, ψ_2 is valid at a time point $t \in [t_1, t_2]$, and $\psi_1 \vee \psi_2$ holds all the time before. (Note that we do not require ψ_1 to hold all the time before, but only the weaker statement $\psi_1 \vee \psi_2$.)
- There is a difference between the CTL and the TCTL semantics of quantification over paths. CTL quantification ranges over all paths. However, timed automata have time-convergent paths that cannot be excluded by modeling. Since those paths are not realistic, they are not considered in the TCTL semantics. Therefore, TCLT quantification ranges over time-divergent paths, only.

Definition 4.10 (Syntax of TCTL). *TCTL* state formulae *over a set AP of atomic propositions and a set $\mathcal{C}$ of clocks can be built according to the abstract grammar*

$$\psi \quad ::= \quad a \mid g \mid (\psi \wedge \psi) \mid (\neg\psi) \mid (E\varphi) \mid (A\varphi)$$

with $a \in AP$, $g \in ACC_{\mathcal{C}}$, and φ are TCTL path formulae. TCTL path formulae *are built according to the abstract grammar*

$$\varphi \quad ::= \quad \psi \, \mathcal{U}^J \, \psi$$

with $J \subseteq \mathbb{R}_{\geq 0}$ is an (open, half-open or closed) interval with integer bounds (open right bound may be ∞), and where ψ are TCTL state formulae. TCTL formulae *are TCTL state formulae.*

Similarly to CTL, we introduce further operators as syntactic sugar. Besides the "finally" and "globally" operators, we consider TCTL formulae with intervals $[0, \infty)$ as CTL formulae.

$$\begin{aligned}
\mathcal{F}^J\psi \quad &:= \quad \mathit{true} \, \mathcal{U}^J \, \psi \\
E\mathcal{G}^J\psi \quad &:= \quad \neg A\mathcal{F}^J\neg\psi \\
A\mathcal{G}^J\psi \quad &:= \quad \neg E\mathcal{F}^J\neg\psi \\
\psi_1 \, \mathcal{U} \, \psi_1 \quad &:= \quad \psi_1 \, \mathcal{U}^{[0,\infty)} \, \psi_2 \\
\mathcal{F}\psi \quad &:= \quad \mathcal{F}^{[0,\infty)}\psi \\
\mathcal{G}\psi \quad &:= \quad \mathcal{G}^{[0,\infty)}\psi
\end{aligned}$$

For the time bounds on temporal operators, we sometimes write $\leq c$, $< c$, ... instead of the intervals $[0, c]$, $[0, c)$, ...

Definition 4.11 (Semantics of TCTL). *Let $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ be a timed automaton, AP a set of atomic propositions, and $L : Loc \to 2^{AP}$ a state labeling function. The satisfaction relation $\models_{TCTL} \subseteq (\Sigma \cup \Pi_{\mathcal{T}}) \times Form_{TCTL}$ (or short $\models$) evaluates TCTL state and path formulae as follows:*

$$
\begin{array}{lll}
\sigma \models_{TCTL} true & & \\
\sigma \models_{TCTL} a & \textit{iff} & a \in L(\sigma) \\
\sigma \models_{TCTL} g & \textit{iff} & \sigma \models_{CC} g \\
\sigma \models_{TCTL} \neg\psi & \textit{iff} & \sigma \not\models_{TCTL} \psi \\
\sigma \models_{TCTL} \psi_1 \wedge \psi_2 & \textit{iff} & \sigma \models_{TCTL} \psi_1 \textit{ and } \sigma \models_{TCTL} \psi_2 \\
 & & \\
\sigma \models_{TCTL} E\varphi & \textit{iff} & \pi \models_{TCTL} \varphi \textit{ for some } \pi \in \Pi_{div}(\sigma) \\
\sigma \models_{TCTL} A\varphi & \textit{iff} & \pi \models_{TCTL} \varphi \textit{ for all } \pi \in \Pi_{div}(\sigma)
\end{array}
$$

where $\sigma \in \Sigma$, $a \in AP$, $g \in ACC(\mathcal{C})$, ψ, ψ_1 and ψ_2 are TCTL state formulae, and φ is a TCTL path formula.
For an infinite path $\pi = \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \sigma_2 \xrightarrow{\alpha_2} \ldots \in \Pi_{div}$ let $d_i = ExecTime(\alpha_i)$. The satisfaction relation for bounded until formulae is defined by

$$
\begin{array}{lll}
\pi \models_{TCTL} \psi_1 \, \mathcal{U}^J \, \psi_2 & \textit{iff} & \textit{there is an } i \geq 0 \textit{ such that } \sigma_i + d \models_{TCTL} \psi_2 \\
 & & \textit{for some } d \in [0, d_i] \textit{ with } (\sum_{k=0}^{i-1} d_k) + d \in J \\
 & & \textit{and for all } j \leq i \textit{ it holds that } \sigma_j + d' \models_{TCTL} \psi_1 \\
 & & \textit{for any } d' \in [0, d_j] \textit{ with either } j < i \textit{ or } d' < d \; .
\end{array}
$$

We define *Sat*

$$Sat(\psi) = \{\sigma \in \Sigma \mid \sigma \models_{TCTL} \psi\}$$

and

$$\mathcal{T} \models_{TCTL} \psi \textit{ iff } \forall \sigma = (\ell, \nu) \in Init \cap Inv(\ell).\ \sigma \models_{TCTL} \psi.$$

Note that TCLT quantification ranges over time-divergent paths, only.

Remark 4.2. The TCTL semantics introduced above is the so-called *continuous* semantics. There is another interpretation of TCTL formulae based on a *pointwise* semantics, the main difference being that along a path $\pi = \sigma_0 \to \sigma_1 \to \ldots$ only the states σ_i are considered in the satisfaction relation but not the other states visited during time steps.

There is also another established variant of the above-defined continuous TCTL semantics, differing in the meaning of the bounded until formula $\psi_1 \, \mathcal{U}^J \, \psi_2$: instead of ψ_1 the weaker requirement $\psi_1 \vee \psi_2$ must hold before the time point of ψ_2.

$$
\begin{array}{lll}
\pi \models_{TCTL} \psi_1 \, \mathcal{U}^J \, \psi_2 & \textit{iff} & \textit{there is an } i \geq 0 \textit{ such that } \sigma_i + d \models_{TCTL} \psi_2 \\
 & & \textit{for some } d \in [0, d_i] \textit{ with } (\sum_{k=0}^{i-1} d_k) + d \in J \\
 & & \textit{and for all } j \leq i \textit{ it holds that } \sigma_j + d' \models_{TCTL} \psi_1 \vee \psi_2 \\
 & & \textit{for any } d' \in [0, d_j] \textit{ with either } j < i \textit{ or } d' < d \; .
\end{array}
$$

4.3 Model Checking TCTL for Timed Automata

After introducing timed automata and the logic TCTL to define properties of timed automata, in this section we give a model checking algorithm to check whether a TCTL formula holds for a given timed automaton. The main problem for model checking TCTL for timed automata lies in the infinite state space. We use abstraction to solve this problem.

The basic structure of the model checking algorithm is as follows:

Input: Non-Zeno timed automaton $\mathcal{T}$ with clock set $\mathcal{C}$,
a labeling function L over a set of atomic propositions AP, and
a TCTL formula ψ over AP and $\mathcal{C}$
Output: The answer to the question whether $\mathcal{T} \models_{TCTL} \psi$

1. Eliminate the timing parameters from ψ, resulting in a formula $\hat{\psi}$ which contains atomic clock constraints but *no intervals* on the temporal operators. If we see atomic clock constraints as atomic propositions then $\hat{\psi}$ is a CTL formula.
2. Make a finite abstraction of the state space, with the abstract states called *regions*.
3. Construct an abstract finite transition system $\mathcal{RTS}$ (*region transition system*) with regions as abstract states, and label the regions with atomic propositions and atomic clock constraints. We have $\mathcal{T} \models_{TCTL} \psi$ iff $\mathcal{RTS} \models_{CTL} \hat{\psi}$.
4. Apply *CTL model checking* to check whether $\mathcal{RTS} \models_{CTL} \hat{\psi}$.
5. *Return* the result of the CTL model checking.

Assume in the following an input for the algorithm in form of a timed automaton $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$, a set of atomic propositions AP, a labeling function $L : Loc \to 2^{AP}$, and a TCTL formula ψ over AP and $\mathcal{C}$.

4.3.1 Eliminating Timing Parameters

Let $\mathcal{T}' = \mathcal{T} \oplus z$ result from $\mathcal{T}$ by adding a fresh clock z which never gets reset. We use this auxiliary clock to measure the time from the beginning of a path and express the time bound of a bounded until as atomic clock constraint. For any state σ of $\mathcal{T}$ it holds that

$$\begin{array}{lll} \sigma \models_{TCTL} E(\psi_1 \, \mathcal{U}^J \, \psi_2) & \textit{iff} & \textit{reset}(z) \textit{ in } \sigma \models_{TCTL} E\psi_1 \, \mathcal{U} \;\; ((z \in J) \wedge \psi_2) \\ \sigma \models_{TCTL} A(\psi_1 \, \mathcal{U}^J \, \psi_2) & \textit{iff} & \textit{reset}(z) \textit{ in } \sigma \models_{TCTL} A\psi_1 \, \mathcal{U} \;\; ((z \in J) \wedge \psi_2) \;\; . \end{array}$$

We transform all subformulae of the TCTL formula ψ to be checked applying the above equivalences, resulting in the formula $\hat{\psi}$. Correctness of the transformation is straightforward for non-nested formulae. For nested formulae we need to slightly adapt the CTL model checking algorithm, as will be explained later (see Section 4.3.4).

Example 4.4. *The TCTL formula* $E\mathcal{F}^{\leq 2} A\mathcal{G}^{[2,3]} a$ *gets transformed into* $E\mathcal{F}(z \leq 2 \wedge A\mathcal{G}(2 \leq z \leq 3 \to a))$.

4.3.2 Finite State Space Abstraction

Since the state space of a timed automaton is in general infinite, to enable model checking we define a *finite abstraction* of the state space. In this abstraction we represent a (possibly infinite) number of states that behave "equivalent" by a single *abstract state*. That two states behave "equivalent" means, that no observation can distinguish between their behavior. Here we do not formalize the notion of *observation* and *observational equivalence*, neither the notion of *bisimulation*. Instead, we define that two states may (but do not have to) be equivalent only if they satisfy the same formulae of a given logic. This definition implies, that model checking the concrete system without abstraction would yield the same result as model checking the abstraction.

Up to the identity relation, an abstraction has in general less states than the concrete system. For this reason, abstraction is widely used also for finite-state systems, since model checking is faster and needs less memory for smaller systems than for larger ones. For infinite-state systems, for which state enumeration is not possible, abstraction may give us a finite-state system which can be model checked.

Before we deal with the abstraction for timed automata and TCTL, let us have a short look at abstractions for the simpler case of labeled state transition systems and the logic CTL*. Assume a labeled state transition system $\mathcal{LSTS}$ with state set Σ, a set of atomic propositions AP, a labeling function $L : \Sigma \to 2^{AP}$, and two states $\sigma_1, \sigma_2 \in \Sigma$. The following conditions assure that σ_1 and σ_2 satisfy the same CTL* formulae:

- To satisfy the same atomic CTL* formulae, i.e., atomic propositions, σ_1 and σ_2 must be labeled with the same set of atomic propositions, i.e., $L(\sigma_1) = L(\sigma_2)$.
- To satisfy the same nested CTL* formulae, for each successor state of σ_1 there must be a successor state of σ_2 such that the two successor states again satisfy the same CTL* formulae, and vice versa, for each successor state of σ_2 there must be a successor state of σ_1 satisfying the same CTL* formulae. Thus we require that if there is a transition from σ_1 to a state σ_1', than there is also a transition from σ_2 to a state σ_2' that is equivalent to σ_1', and vice versa.

Due to this inductive definition, we say that equivalent states can "mimic" each other's behavior in terms of atomic propositions.

The transition system $\mathcal{LSTS}$ may be parallel composed with other LSTSs. In this case label synchronization has to be considered. In order to be able to do the same synchronization steps from equivalent states, we extend the previous requirements as follows (the extensions are emphasized):

- As before, to satisfy the same atomic CTL* formulae, i.e., atomic propositions, σ_1 and σ_2 must be labeled with the same set of atomic propositions, i.e., $L(\sigma_1) = L(\sigma_2)$.
- We require that if there is a transition from σ_1 to a state σ_1' *with label* a, than there is also a transition *with the same label* a from σ_2 to a state σ_2' that is equivalent to σ_1', and vice versa.

We say that equivalent states can "mimic" each other's behavior in terms of atomic propositions *and transition labels*. For a LSTS, a bisimulation is defined to be an equivalence relation on the state set satisfying the above conditions for each pair of equivalent states.

Let us try to extend the above conditions to timed automata and for the logic TCTL. Due to the discrete steps of timed automata, we will need similar conditions as above to cover atomic propositions and discrete steps. However, timed automata has additionally continuous steps, and TCTL may refer to atomic clock constraints. Thus we additionally require that equivalent states can mimic also the time steps of each other, and that equivalent states satisfy, in addition to atomic propositions, also the same atomic clock constraints.

Assume now a timed automaton with state space Σ. Two states $\sigma_1 = (\ell_1, \nu_1) \in \Sigma$ and $\sigma_2 = (\ell_2, \nu_2) \in \Sigma$ are equivalent, implying that they satisfy the same TCTL formulae, if the following conditions hold (the extensions are again emphasized):

- To satisfy the same atomic TCTL formulae, i.e., atomic propositions *and atomic clock constraints*, σ_1 and σ_2 must be labeled with the same set of atomic propositions, i.e., $L(\ell_1) = L(\ell_2)$, and *must satisfy the same atomic clock constraints.*
- We require that if there is a discrete transition from σ_1 to a state σ_1' with label a, than there is also a discrete transition with label a from σ_2 to a state σ_2' that is equivalent to σ_1', and vice versa.

- *For each time step from σ_1 in a successor state σ_1' there is also a time step from σ_2 to some σ_2' such that σ_2' is equivalent to σ_1', and vice versa.*

The above conditions are similar to the definition of *time-abstract bisimulation* (which does not consider atomic clock constraints). Note that for the time steps, the actual duration of the mimicking time step is not important, as long as the successor states cannot be distinguished by any TCTL formulae. This fact will become more clear later, when defining the abstraction for timed automata.

The above conditions would still lead to an infinite abstract state space, since there are infinitely many different clock constraints with different satisfying state sets. However, we need a *finite* abstraction to check *a certain TCTL property*. Consequently, equivalent states do not have to satisfy the same TCTL formulae but only the same subformulae of the given TCTL property. Thus we can release the requirements for *all* clock constraints to clock constraints appearing in the given timed automaton or in the given formula.

Assume a timed automaton $\mathcal{T}$ with locations *Loc*, clocks $\mathcal{C}$, and state space Σ. Assume furthermore an atomic proposition set AP, a labeling function $L : \mathit{Loc} \to 2^{AP}$, and a TCTL formula ψ. Below we define an abstraction by an equivalence relation $\cong \subseteq \Sigma \times \Sigma$ on the states of $\mathcal{T}$. We use

- $\lfloor r \rfloor$ to denote the integral part of $r \in \mathbb{R}$, i.e., $\max\{c \in \mathbb{N} \mid c \leq r\}$, and
- $\mathit{fr}(r)$ to denote the fractional part of $r \in \mathbb{R}$, i.e., $r - \lfloor r \rfloor$.

For clock constraints $x < c$ with $c \in \mathbb{N}$ we have:

$$\nu \models x < c \;\Leftrightarrow\; \nu(x) < c \;\Leftrightarrow\; \lfloor \nu(x) \rfloor < c.$$

For clock constraints $x \leq c$ with $c \in \mathbb{N}$ we have:

$$\nu \models x \leq c \;\Leftrightarrow\; \nu(x) \leq c \;\Leftrightarrow\; \lfloor \nu(x) \rfloor < c \vee (\lfloor \nu(x) \rfloor = c \wedge \mathit{fr}(\nu(x)) = 0)\,.$$

That means, if we would require that equivalent states should satisfy the same clock constraints over the clock set $\mathcal{C}$, then only states (ℓ, ν) and (ℓ, ν') satisfying

$$\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor \text{ and } \mathit{fr}(\nu(x)) = 0 \text{ iff } \mathit{fr}(\nu'(x)) = 0$$

for all $x \in \mathcal{C}$ could be equivalent. However, as mentioned above, if we distinguish between all possible integral parts in $\mathbb{N}$, we would generate infinitely many equivalence classes.

Given the timed automaton $\mathcal{T}$ and the TCTL formula ψ, we are only interested in those clock constraints that play a role in the satisfaction or violation of ψ by $\mathcal{T}$. I.e., it is sufficient if equivalent states satisfy the same clock constraints occurring in $\mathcal{T}$ or ψ.

Let c_x be the largest constant which a clock x is compared to in $\mathcal{T}$ or in ψ. Then there is no observation which could distinguish between the x-values in (ℓ, ν) and (ℓ, ν') if $\nu(x) > c_x$ and $\nu'(x) > c_x$. I.e., equivalent states $(\ell, \nu) \cong (\ell, \nu')$ should satisfy

$$\begin{aligned} &(\nu(x) > c_x \wedge \nu'(x) > c_x) \quad \vee \\ &(\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor \;\wedge\; \mathit{fr}(\nu(x)) = 0 \text{ iff } \mathit{fr}(\nu'(x)) = 0) \end{aligned} \tag{4.1}$$

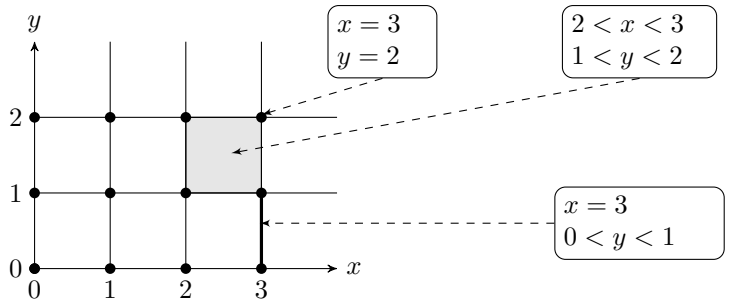
for all $x \in \mathcal{C}$.

Example 4.5. *Assume that $\mathcal{T}$ has two clocks x and y with $c_x = 3$ and $c_y = 2$, i.e., the largest constant that x is compared to in $\mathcal{T}$ or in ψ is 3, and for y this is 2.*

Then we can possibly observe different behavior for states satisfying $x = 0$, $0 < x < 1$, $x = 1$, $1 < x < 2$, $x = 2$, $2 < x < 3$, $x = 3$, and $x > 3$. I.e., two states that satisfy two different clock constraints from the above list must not be equivalent.

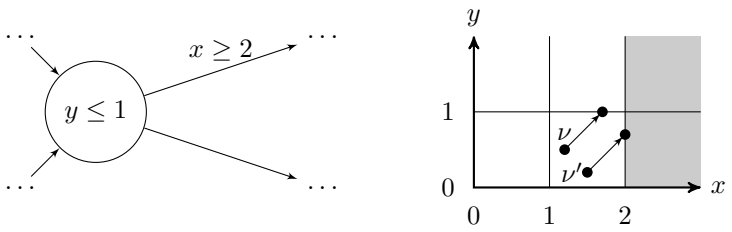
Similarly for y, only states satisfying the same clock constraint from the list $y = 0$, $0 < y < 1$, $y = 1$, $1 < y < 2$, $y = 2$, and $y > 2$ may be equivalent.

In the graphical representation below, valuations belonging to different points, line fragments, or boxes must not be equivalent. This yields at least 48 equivalence classes.



As the following example illustrates, we must make a further refinement of the abstraction.

Example 4.6. *Assume the following fraction of a timed automaton and the corresponding classification of states according to the above observations:*



If the control is in location l with a valuation ν with, e.g., $\nu(x) = 1.2$ and $\nu(y) = 0.5$, then the transition with condition $x \geq 2$ cannot be taken, since the invariant $y \leq 1$ forces the control to leave the location before the value of x reaches 2. But if the valuation assigns, e.g., $\nu'(x) = 1.5$ and $\nu'(y) = 0.2$, then the transition gets enabled before the invariant gets violated.

Though the classification respects Equation 4.1, the valuations in the classes are not yet of the same behavior.

What we need is a refinement taking the order of the fractional parts of the clock values into account. I.e., we must extend the condition of Equation 4.1 with the requirement that states (ℓ, ν) and (ℓ, ν') may be equivalent only if for all clock pairs $x, y \in \mathcal{C}$ with $\nu(x), \nu'(x) \leq c_x \wedge \nu(y), \nu'(y) \leq c_y$

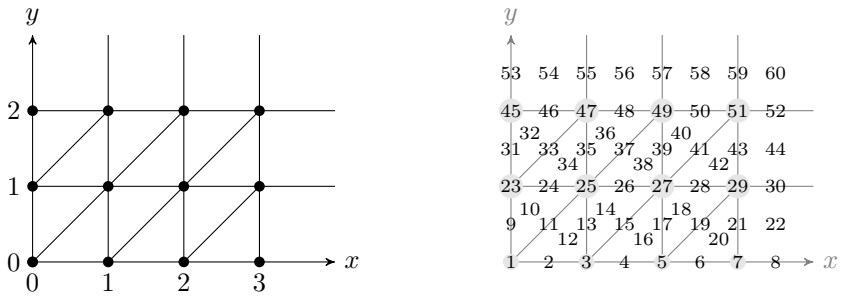
$$\begin{aligned} &fr(\nu(x)) < fr(\nu(y)) \quad \textit{iff} \quad fr(\nu'(x)) < fr(\nu'(y)) \quad \wedge \\ &fr(\nu(x)) = fr(\nu(y)) \quad \textit{iff} \quad fr(\nu'(x)) = fr(\nu'(y)) \quad \wedge \\ &fr(\nu(x)) > fr(\nu(y)) \quad \textit{iff} \quad fr(\nu'(x)) > fr(\nu'(y)). \end{aligned}$$

Because of symmetry requiring

$$fr(\nu(x)) \leq fr(\nu(y)) \quad \textit{iff} \quad fr(\nu'(x)) \leq fr(\nu'(y)). \tag{4.2}$$

is sufficient.

Example 4.7. *We extend the graphical representation of the clock equivalence classes from Example 4.5 taking the conditions of both Equations 4.1 and 4.2 into account. Below, the left picture shows the division of the state space into regions, whereas the right picture enumerates the resulting regions.*



Definition 4.12. *For a timed automaton $\mathcal{T}$ and a TCTL formula ψ, both over a clock set $\mathcal{C}$, we define the* clock equivalence *relation $\cong \subseteq \Sigma \times \Sigma$ by $(\ell, \nu) \cong (\ell', \nu')$ iff $\ell = \ell'$ and*

- *for all $x \in \mathcal{C}$, either $\nu(x) > c_x \wedge \nu'(x) > c_x$ or*

$$\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor \ \wedge \ (fr(\nu(x)) = 0 \quad \textit{iff} \quad fr(\nu'(x)) = 0)$$

- *for all $x, y \in \mathcal{C}$ if $\nu(x), \nu'(x) \leq c_x$ and $\nu(y), \nu'(y) \leq c_y$ then*

$$fr(\nu(x)) \leq fr(\nu(y)) \quad \textit{iff} \quad fr(\nu'(x)) \leq fr(\nu'(y)).$$

The clock region *of an evaluation $\nu \in V$ is the set $[\nu] = \{\nu' \in V \mid \nu \cong \nu'\}$. The* state region *of a state $(\ell, \nu) \in \Sigma$ is the set $[(\ell, \nu)] = \{(\ell, \nu') \in \Sigma \mid \nu \cong \nu'\}$. We also write (ℓ, r) for $\{(\ell, \nu) \mid nu \in r\}$.*

4.3.3 The Region Transition System

After we have defined state regions, next we define how to connect them by abstract transitions, yielding an abstract transition system, which we call the region transition system.

We extend the satisfaction relation for clock constraints to regions by defining

$$\begin{aligned} r \models g \quad & \textit{iff} \quad \forall \nu \in r.\ \nu \models g \\ (\ell, r) \models g \quad & \textit{iff} \quad r \models g. \end{aligned}$$

for r being a clock region of $\mathcal{T}$ with clocks $\mathcal{C}$ and a TCTL formula ψ, and $g \in ACC_{\mathcal{C}} \cup ACC_{\psi}$. On the right-hand side, instead of the universal quantification we could have also required just the existence of a valuation in r satisfying g, as it holds that

$$\forall \nu, \nu' \in r.\ \nu \models g \ \leftrightarrow \ \nu' \models g.$$

We also extend the reset operator to regions as follows:

$$reset(C)\ in\ r \quad = \quad \{(\ell, reset(C)\ in\ \nu) \in \Sigma \mid (\ell, \nu) \in r\}.$$

Note that $reset(C)\ in\ r$ is again a region.

Definition 4.13. *The clock region* $r_\infty = \{\nu \in V \mid \forall x \in \mathcal{C}.\ \nu(x) > c_x\}$ *is called* unbounded. *Let* r, r' *be two clock regions. The region* r' *is the* successor clock region *of* r*, denoted by* $r' = succ(r)$*, if either*

- $r = r' = r_\infty$*, or*
- $r \neq r_\infty$, $r \neq r'$*, and for all* $\nu \in r$*:*

$$\exists d \in \mathbb{R}_{>0}.\ (\nu + d \in r' \ \wedge\ \forall 0 \leq d' \leq d.\ \nu + d' \in r \cup r').$$

The successor state region *is defined as* $succ((\ell, r)) = (\ell, succ(r))$.

Definition 4.14. *Let* $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ *be a non-Zeno timed automaton and let* $\hat{\psi}$ *be an unbounded TCTL formula over* $\mathcal{C}$ *and a set AP of atomic propositions. The region transition system of* $\mathcal{T}$ *for* $\hat{\psi}$ *is a labeled state transition system* $\mathcal{RTS}(\mathcal{T}, \hat{\psi}) = (\Sigma', Lab', Edge', Init')$ *with*

- Σ' *the finite set of all state regions,*
- $Lab' = Lab \cup \{\tau\}$,
- $Init' = \{[\sigma] \mid \sigma \in Init\}$,

and

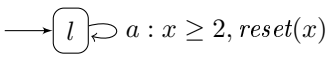
$$\frac{(\ell, a, (g, \mathcal{C}), \ell') \in Edge \qquad r \models g \quad r' = reset(\mathcal{C})\ in\ r \quad r' \models Inv(\ell')}{(\ell, [\nu]) \xrightarrow{a} (\ell', [\nu'])} \quad \textit{Rule}_{\,discrete}$$

$$\frac{r \models Inv(\ell) \quad succ(r) \models Inv(\ell)}{(\ell, r) \xrightarrow{\tau} (\ell, succ(r))} \quad \textit{Rule}_{\,time}$$

Assume a labeling function $L : \Sigma \to 2^{AP}$ *of* $\mathcal{T}$*. We define*

- $AP' = AP \cup ACC(\mathcal{T}) \cup ACC(\psi)$
- $L'((\ell, r)) = L(\ell) \cup \{g \in AP' \backslash AP \mid r \models g\}$

Example 4.8. *Assume the following timed automaton having a single clock* x*:*



Without taking any TCTL formula into account, the abstraction distinguishes the following equivalence classes:

$$\begin{aligned}
r_{[0,0]} &= \{(\ell, \nu) \in \Sigma \mid \nu(x) = 0\} \\
r_{(0,1)} &= \{(\ell, \nu) \in \Sigma \mid 0 < \nu(x) < 1\} \\
r_{[1,1]} &= \{(\ell, \nu) \in \Sigma \mid \nu(x) = 1\} \\
r_{(1,2)} &= \{(\ell, \nu) \in \Sigma \mid 1 < \nu(x) < 2\} \\
r_{[2,2]} &= \{(\ell, \nu) \in \Sigma \mid \nu(x) = 2\} \\
r_{(2,\infty)} &= \{(\ell, \nu) \in \Sigma \mid \nu(x) > 2\}
\end{aligned}$$

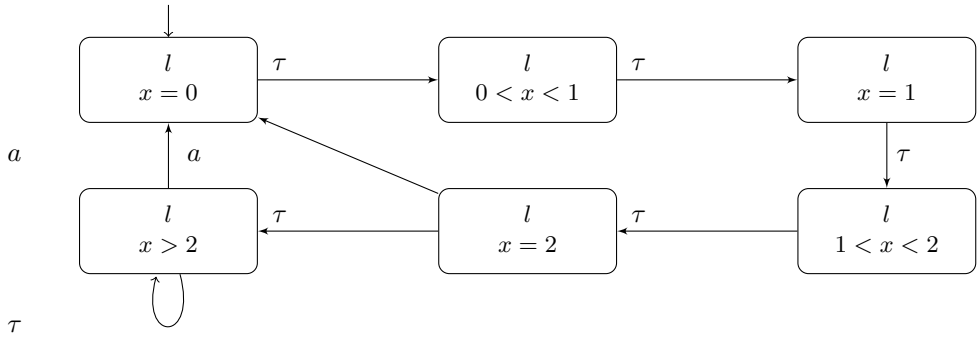
For the transitions, τ*-transitions are defined from each region into its successor region:*

$$\begin{array}{lll}
r_{[0,0]} \xrightarrow{\tau} r_{(0,1)} & r_{(0,1)} \xrightarrow{\tau} r_{[1,1]} & r_{[1,1]} \xrightarrow{\tau} r_{(1,2)} \\
r_{(1,2)} \xrightarrow{\tau} r_{[2,2]} & r_{[2,2]} \xrightarrow{\tau} r_{(2,\infty)} & r_{(2,\infty)} \xrightarrow{\tau} r_{(2,\infty)}
\end{array}$$

Discrete transitions are possible from the regions with $x \geq 2$ into the region with $x = 0$:

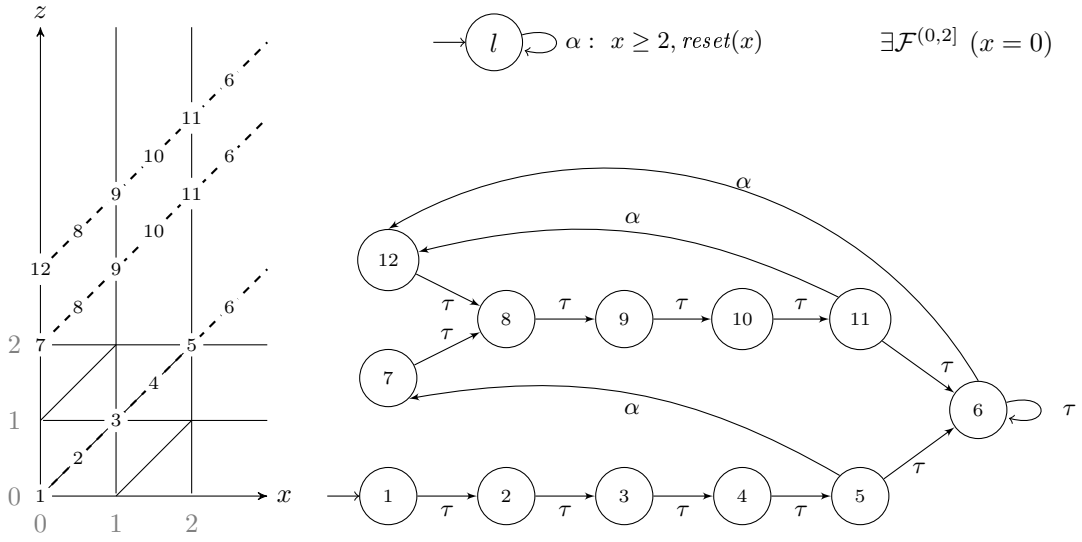
$$r_{[2,2]} \xrightarrow{a} r_{[0,0]} \qquad r_{(2,\infty)} \xrightarrow{a} r_{[0,0]}$$

The resulting region transition graph can be visualized as follows, where for clarity we write into the states the locations and the constraints to which they correspond:

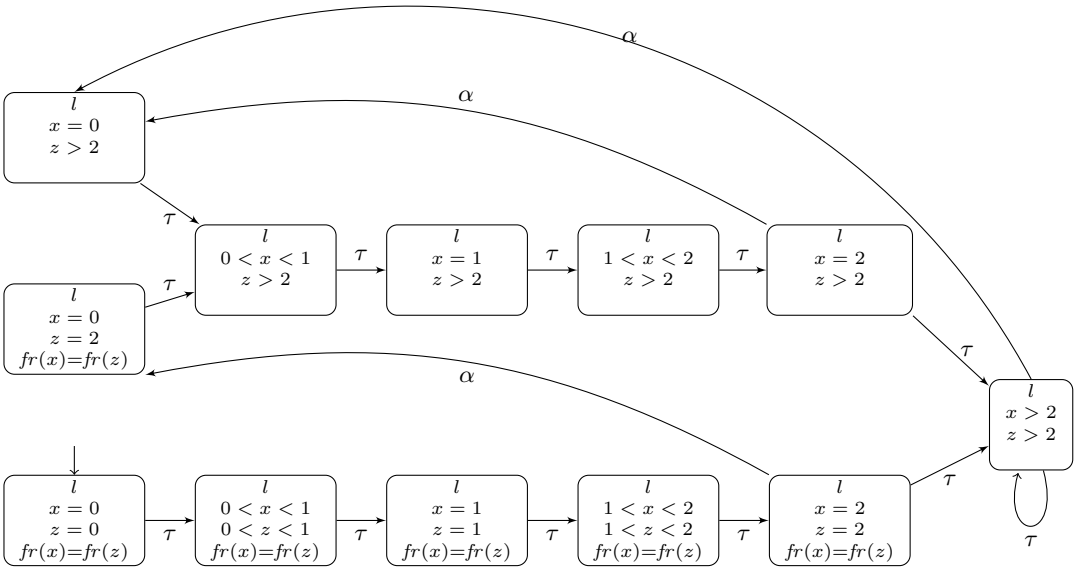


Example 4.9. *Assume the same timed automaton as in the previous Example 4.8, but now additionally consider the TCTL formula $E\mathcal{F}^{(0,2]}(x = 0)$. After removing the bound we get the unbounded formula $E\mathcal{F}(0 < z \leq 2 \wedge x = 0)$. Thus we have $c_x = 2$ and $c_z = 2$.*

We get the following region transition system, where we omit unreachable abstract states. Dotted lines in the coordinate system represent possible behaviors, moving through the different regions.



The following graph shows again the region transition system where the abstract states are annotated with the information determining the regions:



The next lemma states that infinite time-convergent paths of a timed automaton correspond to finite paths in the region transition system.

Lemma 4.1. *For non-Zeno $\mathcal{T}$ and $\pi = s_0 \to s_1 \to \ldots$ an infinite path of $\mathcal{T}$:*

- *if π is time-convergent, then there is an index j and a state region (ℓ, r) such that $s_i \in (\ell, r)$ for all $i \geq j$.*
- *if there is a state region (ℓ, r) with $r \neq r_\infty$ and an index j such that $s_i \in (\ell, r)$ for all $i \geq j$ then π is time-convergent.*

Theorem 4.2. *A non-Zeno timed automaton $\mathcal{T}$ is timelock free iff its region transition system does not have any deadlocks, i.e., reachable terminal states.*

4.3.4 TCTL Model Checking

The procedure is quite similar to CTL model checking for finite automata. The only difference concerns the handling of nested time bounds in TCTL formulae.

As in CTL model checking, we label the abstract states of the region transition system with subformulae of the formula ψ to be checked, inside-out starting with the inner-most subformulae. However, since we want to use a single auxiliary clock, we must additionally represent the "restart" of the auxiliary clock at some places.

To explain the problem, consider the formula $E\mathcal{F}^{[0,1]}(a \wedge E\mathcal{F}^{[1,2]}b)$. Removing the bounds yields $E\mathcal{F}(0 \leq z \leq 1 \wedge a \wedge E\mathcal{F}(1 \leq z \leq 2 \wedge b))$. The labeling with the atomic propositions a and b is defined by the labeling function. The labeling with atomic clock constraints is done upon the generation of the region transition system. The first step of the model checking algorithm would label those regions with $1 \leq z \leq 2 \wedge b$ that are labeled with $1 \leq z$, $z \leq 2$, and b. Now we come to the more interesting part: the algorithm would determine all those regions from which a region labeled with $1 \leq z \leq 2 \wedge b$ is reachable, and may label them with $E\mathcal{F}(1 \leq z \leq 2 \wedge b)$. Now we

make two observations: Firstly, $E\mathcal{F}^{[1,2]}b$ is satisfied only by those determined regions that are labeled with $z = 0$. Secondly, the start value $z = 0$ of the auxiliary clock is just a convention, we could also have started with a value, e.g., $z = 2$ and check reachability of $3 \leq z \leq 4 \wedge b$. Consequently, we should label all those regions r with $E\mathcal{F}^{[1,2]}b$ for that the region $reset(z)$ *in* r is labeled with $E\mathcal{F}(1 \leq z \leq 2 \wedge b)$. The labeling for the other subformulae is analogous. After termination, the timed automaton satisfies the above TCTL formula iff each initial region is labeled with it.

Lemma 4.2. *For a non-Zeno timed automaton $\mathcal{T}$ and an unbounded TCTL formula ψ:*

$$\mathcal{T} \models_{TCTL} \psi \quad \textit{iff} \quad \mathcal{RTS}(\mathcal{T}, \psi) \models_{CTL} \hat{\psi}$$

Lemma 4.3. *The model checking problem for timed automata and TCTL properties is complete for PSPACE.*

Chapter 5

Rectangular Automata

In the previous chapter we have seen that TCTL for timed automata, a special class of hybrid automata, is decidable, thus model checking is possible. In this chapter we discuss a bit more general class, the class of *rectangular automata*, and analyse decidability. The contents of this chapter are based on [HKPV98].

Rectangular automata build an interesting class of hybrid automata because on the one hand they allow a more expressive modeling than timed automata and on the other hand (under some additional conditions) both safety and liveness for rectangular automata are decidable. However, they lie on the boundary of decidability in the sense that several slight generalizations lead to undecidability.

In the previous chapters we used temporal logics supporting the specification of both safety and liveness properties. From now on we restrict ourselves to *safety* properties, stating that each reachable state of an automaton is included in a given set of safe states.

In the following Section 5.1 we define syntax and semantics of rectangular automata, before discussing decidability in Section 5.2.

5.1 Syntax and Semantics

In the following we first formally define the syntax and semantics of rectangular automata. As rectangular automata are special hybrid automata, their states $\sigma = (l, \nu) \in \Sigma = Loc \times V$ also consist of a discrete component describing the current location, and of a valuation component, assigning values to the real-valued variables. To simplify the notation, in the following we assume that the real-valued variables $Var = \{x_1, \ldots, x_d\}$ of the automata are ordered and write $(l, v) \in Loc \times \mathbb{R}^d$ for a state (l, ν) with $\nu(x_i) = v_i$ for all $i = 1, \ldots, d$.

To define rectangular automata we first need to define rectangular sets.

Definition 5.1 (Rectangular set). *A set $\mathcal{R} \subset \mathbb{R}^d$ is* rectangular *if it is a cartesian product of (possibly unbounded) intervals, all of whose finite endpoints are rational. The set of rectangular sets in $\mathbb{R}^d$ is denoted by $\mathcal{R}^d$.*
Given a set Loc of locations, a subset of the state space $Loc \times \mathbb{R}^d$ is called a zone. *Each zone Z is decomposable into a collection $\bigcup_{l \in Loc} \{l\} \times Z_l$ of zones. The zone Z is* rectangular *iff each Z_l is rectangular. A zone is* multirectangular, *if it is a finite union of rectangular zones.*

Rectangular automata are hybrid automata whose invariants, activities, and transition relations are all described by rectangular sets. For the invariants and transition guards it means that

those conditions may not compare the values of different variables to each other, but to constant values only. Similarly, a transition may reset the value of a variable to a non-deterministically chosen value from an interval, whose end-points are constants, i.e., they do not depend on the values of other variables. Finally, the activities assign constant lower and upper bounds to the derivatives, allowing also non-linear behaviour. However, since the evolution of a variable may not depend on the value of another variable, the set of states reachable via time steps from a rectangular set is again a rectangular set.

Definition 5.2 (Syntax of rectangular automata). *A* d-dimensional rectangular automaton *(or short rectangular automaton) is a tuple* $\mathcal{H} = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ *with*

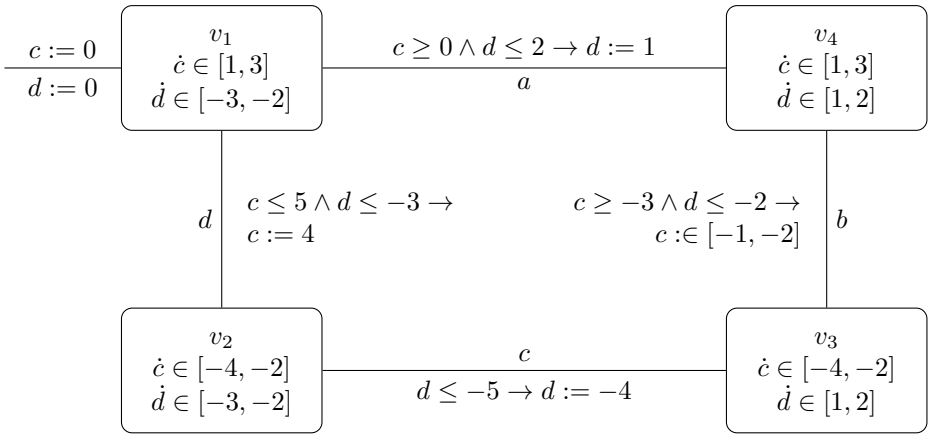
- *a finite set* Loc *of locations;*
- *a finite set* $Var = \{x_1, \ldots, x_d\}$ *of* d *ordered real-valued variables; we write* $x = (x_1, \ldots, x_d)$ *for the ordered sequence of the variables;*
- *a function* $Con : Loc \to 2^{Var}$ *assigning a set of controlled variables to each location;*
- *a finite set* Lab *of synchronization labels;*
- *a set* $Edge \subseteq Loc \times Lab \times (\mathcal{R}^d \times \mathcal{R}^d \times 2^{\{1,\ldots,n\}}) \times Loc$ *of edges;*
- *a flow function* $Act : Loc \to \mathcal{R}^d$*;*
- *an invariant function* $Inv : Loc \to \mathcal{R}^d$*;*
- *initial states* $Init : Loc \to \mathcal{R}^d$*.*

A rectangular automaton is initialized *iff for all edges* $e = (l, a, pre, post, jump, l') \in Edge$ *and all* $i \in \{1, \ldots, n\}$ *we have that if* $Act(l)_i \neq 0$ *and* $Act(l)_i \neq Act(l')_i$ *then* $i \in jump$*, where* $Act(l)_i$ *is the projection of* $Act(l)$ *to the* i*th dimension.*

For the flows, the first time derivatives of the flow trajectories in location $l \in Loc$ are within the rectangular set $Act(l)$. For the jumps, an edge $e = (l, a, pre, post, jump, l') \in Edge$ may move control from location l to location l' starting from a valuation in pre, changing the value of each variable $x_i \in jump$ to a nondeterministically chosen value from $post_i$ (the projection of $post$ to the ith dimension), and leaving the values of the other variables unchanged.

An initialized rectangular automaton has the property that whenever the flow of a variable changes due to a discrete transition, the variable is re-initialized to a value from an interval with constant bounds. The reachability problem for initialized rectangular automata is decidable. However, it becomes undecidable if the restriction of being initialized is relaxed.

Example 5.1. *The following graph illustrates is an initialized rectangular automaton:*



Note that a timed automaton is a special rectangular automaton such that every variable is a clock, the initial sets $Init(l)$ are empty or are singletons for each location $l \in Loc$, and the edges reset variables to 0 only. Furthermore, if we replace rectangular regions with linear regions, we obtain linear hybrid automata, a super-class of rectangular automata, which are the subject of the next chapter.

The semantics of rectangular automata is derived from the semantics of hybrid automata as follows.

Definition 5.3 (Semantics of rectangular automata). *The* operational semantics *of a rectangular automaton $\mathcal{H} = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ is given by the following two rules:*

$$\frac{(l, a, pre, post, jump, l') \in Edge \qquad v \in pre \quad v' \in post \quad \forall i \notin jump. v'_i = v_i \quad v' \in Inv(l')}{(l, v) \xrightarrow{a} (l', v')} \quad Rule_{discrete}$$

$$\frac{(t = 0 \wedge v = v') \vee (t > 0 \wedge (v' - v)/t \in Act(l)) \quad v' \in Inv(l)}{(l, v) \xrightarrow{t} (l, v')} \quad Rule_{time}$$

The one-step transition *is given by* $\rightarrow = \xrightarrow{a} \cup \xrightarrow{t}$, *its* transitive closure *by* $\rightarrow^*$. *A* path *is a sequence* $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \ldots$. *starting in an initial state* $\sigma_0 = (\ell_0, v_0)$ *with* $v_0 \in Init(\ell_0) \cap Inv(\ell_0)$. *A state is* reachable *iff there exists a path leading to it.*

Note that, similarly to timed automata, the invariant sets of rectangular automata are convex. Furthermore, though the time behaviour can be non-linear, for each non-linear time flow there is a corresponding linear one leading to the same state in the same time. Thus for the time steps we do not need to require the invariant to hold at each time point during the time step, but it is sufficient to require that the invariant holds initially and after each step.

Lemma 5.1. *For every multirectangular zone Z of a d-dimensional rectangular automaton $\mathcal{H}$, and every label $lab \in Lab \cup \mathbb{R}_{\geq 0}$, the zones $Post^{lab}(Z) = \{(l', v') \in Loc \times \mathbb{R}^d \mid \exists (l, v) \in Z.\ (l, v) \xrightarrow{lab} (l', v')\}$ and $Pre^{lab}(Z) = \{(l, v) \in Loc \times \mathbb{R}^d \mid \exists (l', v') \in Z.\ (l, v) \xrightarrow{lab} (l', v')\}$ are multirectangular.*

Proof. It suffices to prove the lemma for elementary regions of the form $Z = (\{l\}, \mathcal{R})$ with $\mathcal{R}$ rectangular. We distinguish between discrete and time steps.

For discrete steps assume $lab = a \in Lab$. Let furthermore $e = (l, a, pre, post, jump, l')$ be an edge. Then $Post^a(Z) = \{l'\} \times S$ with

$$S_i = \begin{cases} \mathcal{R}_i \cap pre_i \cap post_i \cap Inv(l')_i & \text{if } i \notin jump, \\ post_i \cap Inv(l')_i & \text{if } i \in jump \text{ and } \mathcal{R}_i \cap pre_i \neq 0, \\ \emptyset & \text{if } i \in jump \text{ and } \mathcal{R}_i \cap pre_i = 0. \end{cases}$$

Thus $Post^a(Z)$ is rectangular, and the union over all edges starting in l with label a is a multirectangular zone.

For time steps, if $lab = 0$ then $Post^0(Z) = Z$. Thus assume $lab = t \in \mathbb{R}$ with $t > 0$. Let $L = inf(\mathcal{R}_i) + t \cdot inf(Act(l)_i)$ and $U = sup(\mathcal{R}_i) + t \cdot sup(Act(l)_i)$.

Then $Post^t(Z) = \{l\} \times S$ with

$$S_i = \begin{cases} Inv(l)_i \cap [L, \infty) \cap (-\infty, U] & \text{if } \mathcal{R}_i \text{ and } Act(l)_i \text{ are closed,} \\ Inv(l)_i \cap (L, \infty) \cap (-\infty, U] & \text{if } \mathcal{R}_i \text{ or } Act(l)_i \text{are left-open and} \\ & \text{both are right-closed, and} \\ Inv(l)_i \cap [L, \infty) \cap (-\infty, U) & \text{if } \mathcal{R}_i \text{ or } Act(l)_i \text{are right-open and} \\ & \text{both are left-closed.} \end{cases}$$

Thus $Post^t(Z)$ is a rectangular zone. □ □

Note that the reachable zone of a rectangular automaton is in general an infinite union of rectangular zones, and may thus be not multirectangular.

5.2 Decidability of Rectangular Automata

The reachability problem for initialized rectangular automata is decidable.

Lemma 5.2. *The reachability problem for initialized rectangular automata is PSPACE complete.*

The proof makes use of the fact that the reachability problem for timed automata is complete for PSPACE. It defines a (polynomial) transformation of initialized rectangular automata to timed automata thereby proving PSPACE completeness. The transformation is done in three steps:

Timed automaton
↑
Initialized stopwatch automaton
↑
Initialized singular automaton
↑
Initialized rectangular automaton

In the following we describe these steps. Note that the transformation does not only prove decidability, but also gives us a model checking algorithm for initialized rectangular automata, since we can apply the previously discussed model checking algorithm to the resulting timed automaton.

5.2.1 From Initialized Stopwatch Automata to Timed Automata

Let us start with the first step transforming an initialized stopwatch automaton into a timed automaton.

Definition 5.4.
- *A rectangular automaton has* deterministic jumps, *if (1) Init*(l) *is empty or a singleton for all* l*, and (2) the post-interval for each variable from the jump-set of each edge is a singleton.*
- *A* stopwatch *is a variable with derivatives* 0 *or* 1 *only.*
- *A* stopwatch automaton *is a rectangular automaton with deterministic jumps and stopwatch variables only.*

Initialized stopwatch automata can be polynomially encoded by timed automata, as shown below. This implies the decidability of initialized stopwatch automata. However, the reachability problem for non-initialized stopwatch automata is undecidable.

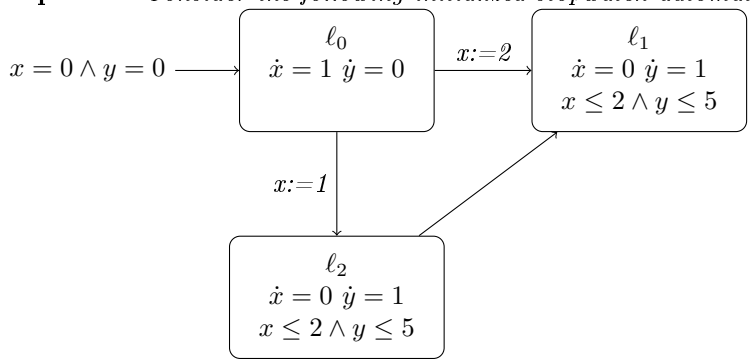
Lemma 5.3. *The reachability problem for initialized stopwatch automata is PSPACE complete.*

The encoding works as follows. First notice, that a timed automaton is a stopwatch automaton such that every variable is a clock.

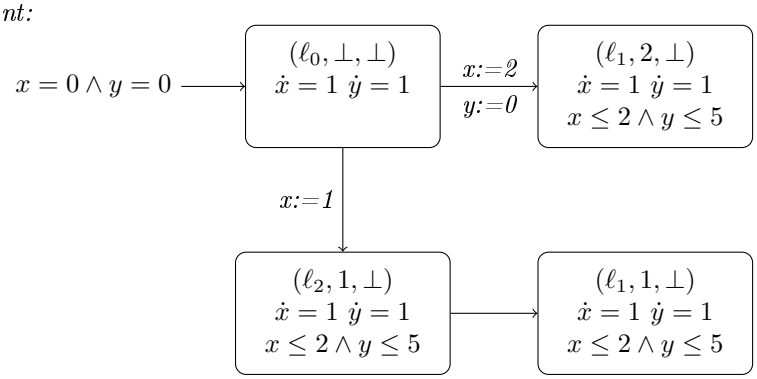
Assume that $\mathcal{H} = (Loc, Var, Con, Lab, Edge, Act, Inv, Init)$ is a d-dimensional (i.e. $|Var| = d$) initialized stopwatch automaton. Let κ be the set of rational constants used in the definition of $\mathcal{H}$, and let $\kappa_\perp = \kappa \cup \{\perp\}$.

Intuitively, if the ith stopwatch of $\mathcal{H}$ is running (slope 1), then its value is tracked by the value of the ith clock of $\mathcal{H}'$; if the ith stopwatch is halted (slope 0) at value $k_i \in \kappa$, then this value is remembered by the current location of $\mathcal{H}'$.

Example 5.2. *Consider the following initialized stopwatch automaton:*



This automaton can be transformed to a timed automaton with the following reachable fragment:



5.2.2 From Initialized Singular Automata to Initialized Stopwatch Automata

Definition 5.5.

- *A variable x_i is a* finite-slope variable *if $flow(l)_i$ is a singleton in all locations l.*
- *A* singular automaton *is a rectangular automaton with deterministic jumps such that every variable of the automaton is a finite-slope variable.*

Lemma 5.4. *The reachability problem for initialized singular automata is PSPACE complete.*

The proof is again based on automata transformation. Initialized singular automata can be rescaled to initialized stopwatch automata as follows.

Let B be a d-dimensional initialized singular automaton with ϵ-moves. We define a d-dimensional initialized stopwatch automaton C_B with the same location set, edge set, and label set as B.

Each state $q = (l, v)$ of C_B corresponds to the state $\beta(q) = (l, \beta(v))$ of B with $\beta : \mathbb{R}^d \to \mathbb{R}^d$ defined as follows:

For each location l of B, if $Act_B(l) = \Pi_{i=1}^{d}[k_i, k_i]$, then $\beta(v_1, \ldots, v_d) = (\ell_1 \cdot v_1, \ldots, \ell_d \cdot v_d)$ with $\ell_i = k_i$ if $k_i \neq 0$, and $\ell_i = 1$ if $k_i = 0$;

β can be viewed as a rescaling of the state space. All conditions in the automaton B occur accordingly rescaled in C_B.

The reachable set $Reach(B)$ of B is $\beta(Reach(C_B))$.

5.2.3 From Initialized Rectangular Automaton to Initialized Singular Automaton

Lemma 5.5. *The reachability problem for initialized rectangular automata is PSPACE complete.*

The proof is based on the translation of a d-dimensional initialized rectangular automaton $\mathcal{H}$ into a $(2n+1)$-dimensional initialized singular automaton B, such that B contains all reachability information about $\mathcal{H}$.

The translation is similar to the subset construction for determinizing finite automata.

The idea is to replace each variable c of $\mathcal{H}$ by two finite-slope variables c_l and c_u: c_l tracks the least possible value of c, and c_u tracks the greatest possible value of c.

Chapter 6

Linear Hybrid Automata I

In this chapter we discuss a further class of hybrid automata called *linear hybrid automata I.* Linear hybrid automata I are time-deterministic hybrid automata whose definitions contain linear terms, only. They are more expressive than timed or rectangular automata, and the reachability problem for linear hybrid automata I is in general undecidable. However, *bounded* reachability, i.e., reachability within a fixed number of steps, is still decidable and can be efficiently computed. Approximation and minimization techniques can be additionally used for the successful analysis of linear hybrid automata I.

We introduce linear hybrid automata I in Section 6.1. Forward and backward analysis techniques are discussed in the Sections 6.2 and 6.3, respectively. Approximation methods for linear hybrid automata I are described in Section 6.4, and we handle minimization in Section 6.5

The contents of this chapter are based on [ACH+95].

6.1 Syntax and Semantics

Definition 6.1.
- *A* linear term *over the set Var of variables is a linear combination of variables in Var with integer (rational) coefficients.*
- *A* linear formula *over Var is a Boolean combination of (in)equalities between linear terms over Var.*
- *A hybrid automaton is* time deterministic *iff for every location $l \in Loc$ and every valuation $\nu \in V$ there is at most one activity $f \in Act(l)$ with $f(0) = \nu$. The activity f, then, is denoted by $f_l[\nu]$, its component for $x \in Var$ by $f_l^x[\nu]$.* $ff_l[\nu]$

The restrictions on the syntax of linear hybrid automata I affect the activities, the invariants, and the discrete edges.

Definition 6.2 (Syntax of linear hybrid automata I). *A* linear hybrid automaton I *is a time-deterministic hybrid automaton with the following properties:*

- Intial states $Init(l)$ *are described by the linear formula* $\varphi_{Init}^{l'}$*, where*

$$Init(l) = \{(\nu, l') | l = l' \wedge \varphi_{Init}^{l'}[\nu(x)/x]\}$$

- Activities $Act(l)$ *follow derivatives* $\dot{x} \in [L - x, u_x]$ *for each variable* $x \in Var$*, with* l_x, u_x *integer (rational) constants. They are described by a linear formula*

$$\varphi_{Act}^{l} = \bigwedge_{x \in Var} x + l_x \cdot t \leq x' \wedge x' \leq x + u_x \cdot t$$

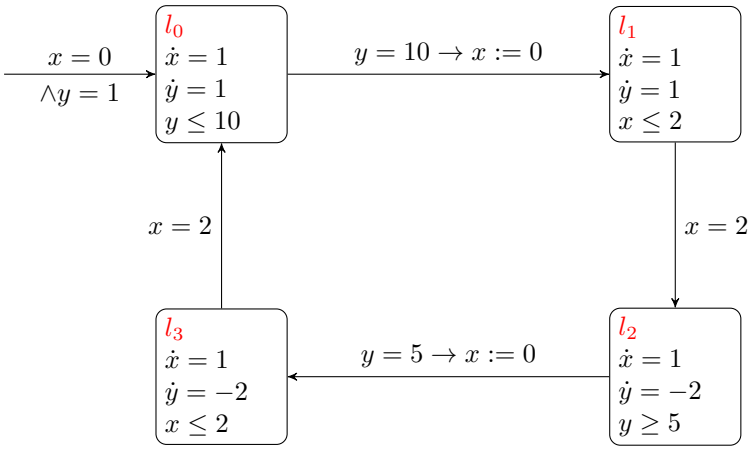


Figure 6.1: Water-level monitor

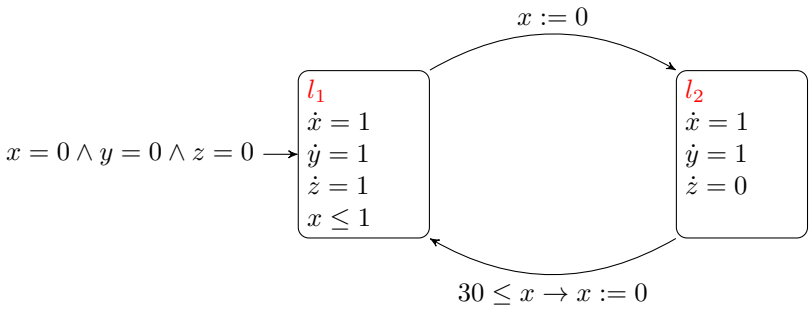


Figure 6.2: Leaking gas burner

such that

$$Act(l) = \{f | \forall t \in \mathbb{R}^{\geq 0}.\varphi^l_{Act}[f(0), f(t)/x, x']\}\}$$

- Invariants $Inv(l)$ *are defined by linear formulae* φ^l_{Inv} *over* Var*:*

$$Inv(l) = \{\nu | \varphi^l_{Inv}[\nu(x)/x]\}$$

- *For all edges, the* transision relation *is defined by a guarded set of nondeterministic assignments:*

$$e = (l, \mu, l')$$

defining

$$\mu = \{(\nu, \nu') | \varphi_{e,guard}[\nu(x)/x] \wedge \varphi_{e,reset}[\nu(x), \nu'(x)/x, x']\}$$

$$\varphi_{e,reset} = \bigwedge_{x \in Var} e^{lower}_x \leq x' \wedge x' \leq e^{upper}_x$$

where the guard $\varphi_{e,guard}$ *is a linear formula.*

Figures 6.1 and 6.2 give two examples for linear hybrid automata I.

$$0 < l_1 < u_1,\ 0 < l_2 < u_2,\ l \leq u < 0,\ d > 0,\ c > 0$$

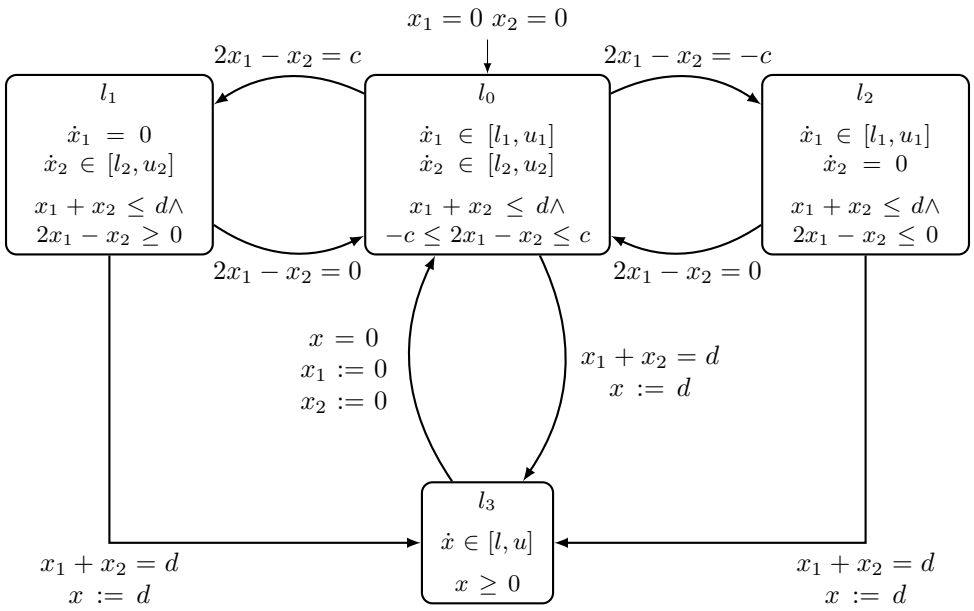


Figure 6.3: Mixer of fluids

The semantics of linear hybrid automata I is given by the semantics of hybrid automata, specified by the following rules for discrete and time steps:

$$\frac{(l, a, \mu, l') \in \mathit{Edge} \quad (\nu, \nu') \in \mu \quad \nu' \in \mathit{Inv}(l')}{(l, \nu) \xrightarrow{a} (l', \nu')}\ \texttt{Rule}_{\texttt{discrete}}$$

$$\frac{\begin{array}{l} f \in \mathit{Act}(l) \quad f(0) = \nu \quad f(t) = \nu' \\ t \geq 0 \quad \forall 0 \leq t' \leq t. f(t') \in \mathit{Inv}(l) \end{array}}{(l, \nu) \xrightarrow{t} (l, \nu')}\ \texttt{Rule}_{\texttt{time}}$$

For time-deterministic hybrid automata the time-step rule can be simplified using the following predicate.

Definition 6.3. *For time-deterministic hybrid automata we define the* "time can progress" *predicate:* $\mathit{tcp}_l[\nu](t)$

$$\mathit{tcp}_l[\nu](t) \quad \textit{iff} \quad \forall 0 \leq t' \leq t.\ f_l[\nu](t') \in \mathit{Inv}(l).$$

Thus for time-deterministic automata we can rewrite the time-step rule to

$$\frac{t \geq 0 \quad \mathit{tcp}_l[\nu](t)}{(l, \nu) \xrightarrow{t} (l, f_l[\nu](t))}\ \texttt{Rule}'_{\texttt{time}}$$

6.2 Forward Analysis

The reachability problem for linear hybrid automata I is in general undecidable. However, bounded reachability is still decidable. Despite of undecidability, for the general rechability analysis of linear hybrid automata I there exist incomplete algorithms. In this section we describe such a technique, a forward analysis approach based on fixed-point computation.

In general, forward analysis techniques start from the initial state set R_0 of a system, and compute the state set R_1 reachable from R_0 within one computation step. For the resulting set the same computation is repeated, i.e., the state set R_2 reachable in one transition step from R_1 is computed. The algorithm terminates if after a number of steps no new states can be reached, i.e., if $R_k \subseteq \bigcup_{i=0}^{k-1} R_i$ for some $k > 0$. Termination corresponds to finding the least fixed-point for the one-step (forward) reachability starting from the initial set. After termination we can check if all states in the determined reachable set satisfy the required property. Note that the computation may in general not terminate if the state space is infinite.

The one-step reachability for continuous steps is described by the following notion of forward time closure:

$\langle P \rangle_l^\nearrow$ **Definition 6.4.** *We define the* forward time closure $\langle P \rangle_l^\nearrow$ *of $P \subseteq V$ at $l \in Loc$ as the set of valuations reachable from P by letting time progress:*

$$\nu' \in \langle P \rangle_l^\nearrow \quad \textit{iff} \quad \exists \nu \in P.\ \exists t \in \mathbb{R}_{\geq 0}.\ tcp_l[\nu](t) \wedge \nu' = f_l[\nu](t).$$

$\langle R \rangle_l^\nearrow$ *We extend the definition to regions $R = \cup_{l \in Loc}(l, R_l)$ as follows:*

$$\langle R \rangle^\nearrow = \cup_{l \in Loc}(l, \langle R_l \rangle_l^\nearrow).$$

For the discrete steps, the corresponding one-step relation is formalized by postconditions:

$post_e[P]$ *We define the* postcondition $post_e[P]$ *of P with respect to an edge $e = (l, a, \mu, l')$ as the set of valuations reachable from P by e:*

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

$post[R]$ *An extension to regions $R = \cup_{l \in Loc}(l, R_l)$ is defined as follows:*

$$post[R] = \cup_{e=(l,a,\mu,l') \in Edge}(l', post_e[R_l]).$$

Note that, due to the τ-transitions, $R \subseteq post[R]$. Similarly, due to time steps of duration 0 we have $R \subseteq \langle R \rangle^\nearrow$.

Lemma 6.1. *For all linear hybrid automata I, if $P \subseteq V$ is a linear set of valuations, then for all $l \in Loc$ and $e \in Edge$, both $\langle P \rangle_l^\nearrow$ and $post_e[P]$ are linear sets of valuations.*

The set of states reachable in a finite number of steps from the initial state set form the reachable region of the automaton.

$(I, \mapsto^*)$ **Definition 6.5.** *Given a region $I \subseteq \Sigma$, the* reachable region $(I \mapsto^*) \subseteq \Sigma$ *of I is the set of all states that are reachable from states in I:*

$$\sigma \in (I \mapsto^*) \quad \textit{iff} \quad \exists \sigma' \in I.\ \sigma' \to^* \sigma.$$

The following lemma states, that if the forward analysis procedure terminates, then the result, being the least fixed-point of the one-step relation, gives us the set of all reachable states.

Lemma 6.2. *Let $I = \cup_{l \in Loc}(l, I_l)$ be a region of the linear hybrid automaton I A. The reachable region $(I, \mapsto^*) = \cup_{l \in Loc}(l, R_l)$ is the least fixed-point of the equation*

$$X = \langle I \cup post[X] \rangle^{\nearrow}$$

or, equivalently, for all locations $l \in Loc$, the set R_l of valuations is the least fixed-point of the set of equations

$$X_l = \langle I_l \cup \bigcup_{e=(l',a,\mu,l) \in Edge} post_e[X_{l'}] \rangle_l^{\nearrow}.$$

Example 6.1 (Example forward reachability computation). *Consider the example automaton from Figure ?? and assume that bad states (l, ν) are characterized by $\nu(x) = \nu(y) + 2$, independently of the location. We represent the initial sets, activities, invariants, transition relations and the bad states by linear real arithmetic formulas as follows:*

$$\begin{array}{lcllcl}
Init_{\ell_1}(x,y) & = & x = 0 \wedge y = 0 & Init_{\ell_2}(x,y) & = & false \\
f^x_{\ell_1}(x,y)(t) & = & x + t & f^x_{\ell_2}(x,y)(t) & = & x \\
f^y_{\ell_1}(x,y)(t) & = & y & f^y_{\ell_2}(x,y)(t) & = & y + t \\
Inv_{\ell_1}(x,y) & = & x \leq y + 1 & Inv_{\ell_2}(x,y) & = & y \leq x + 1 \\
 & & & & & \\
\mu^x_{\ell_2 \to \ell_1}(x,y) & = & x & \mu^x_{\ell_1 \to \ell_2}(x,y) & = & x \\
\mu^y_{\ell_2 \to \ell_1}(x,y) & = & y & \mu^y_{\ell_1 \to \ell_2}(x,y) & = & y \\
 & & & & & \\
Bad_{\ell_1}(x,y) & = & x = y + 2 & Bad_{\ell_2}(x,y) & = & x = y + 2
\end{array}$$

The forward reachability analysis computes the following state sets represented again as linear real arithmetic formulas:

$$\begin{array}{lcl}
R^0_{\ell_1}(x,y) & = & Init_{\ell_1}(x,y) \wedge Inv_{\ell_1}(x,y) = x = 0 \wedge y = 0 \wedge x \leq y + 1 \\
R^0_{\ell_2}(x,y) & = & Init_{\ell_2}(x,y) \wedge Inv_{\ell_2}(x,y) = false \\
 & & \\
R^1_{\ell_1}(x,y) & = & \mathcal{T}^+_{\ell_1}(R^0_{\ell_1}) \\
 & = & \exists x', y', t.\ R^0_{\ell_1}(x',y') \wedge t \geq 0 \wedge x = f^x_{\ell_1}(x',y')(t) \wedge y = f^y_{\ell_1}(x',y')(t) \wedge Inv_{\ell_1}(x,y) \\
 & = & \exists x', y', t.\ \underbrace{x' = 0}_{elim.\ x'} \wedge \underbrace{y' = 0}_{elim.\ y'} \wedge x' \leq y' + 1 \wedge t \geq 0 \wedge x = x' + t \wedge y = y' \wedge x \leq y + 1 \\
 & = & \exists t.\ 0 \leq 1 \wedge t \geq 0 \wedge \underbrace{x = t}_{elim.\ t} \wedge y = 0 \wedge x \leq y + 1 \\
 & = & x \geq 0 \wedge y = 0 \wedge x \leq y + 1 \\
R^1_{\ell_2}(x,y) & = & \mathcal{T}^+_{\ell_2}(R^0_{\ell_2}) \\
 & = & \exists x', y', t.\ R^0_{\ell_2}(x',y') \wedge t \geq 0 \wedge x = f^x_{\ell_2}(x',y')(t) \wedge y = f^y_{\ell_2}(x',y')(t) \wedge Inv_{\ell_2}(x,y) \\
 & = & false \\
 & & \\
R^2_{\ell_1}(x,y) & = & \mathcal{D}^+(R^1_{\ell_2}) \\
 & = & \exists x', y'.\ R^1_{\ell_2}(x',y') \wedge x = \mu^x_{\ell_2 \to \ell_1}(x',y') \wedge y = \mu^y_{\ell_2 \to \ell_1}(x',y') \wedge Inv_{\ell_1}(x,y) \\
 & = & false \\
R^2_{\ell_2}(x,y) & = & \mathcal{D}^+(R^1_{\ell_1}) \\
 & = & \exists x', y'.\ R^1_{\ell_1}(x',y') \wedge x = \mu^x_{\ell_1 \to \ell_2}(x',y') \wedge y = \mu^y_{\ell_1 \to \ell_2}(x',y') \wedge Inv_{\ell_2}(x,y) \\
 & = & \exists x', y'.\ x' \geq 0 \wedge y' = 0 \wedge x' \leq y' + 1 \wedge \underbrace{x = x'}_{elim.\ x'} \wedge \underbrace{y = y'}_{elim.\ y'} \wedge y \leq x + 1 \\
 & = & x \geq 0 \wedge y = 0 \wedge x \leq y + 1 \wedge y \leq x + 1
\end{array}$$

$$
\begin{array}{lcl}
R^3_{\ell_1}(x,y) & = & \mathcal{T}^+_{\ell_1}(R^2_{\ell_1}) \\
& = & \exists x',y',t.\ R^2_{\ell_1}(x',y') \wedge t \geq 0 \wedge x = f^x_{\ell_1}(x',y')(t) \wedge y = f^y_{\ell_1}(x',y')(t) \wedge Inv_{\ell_1}(x,y) \\
& = & \mathit{false} \\
R^3_{\ell_2}(x,y) & = & \mathcal{T}^+_{\ell_2}(R^2_{\ell_2}) \\
& = & \exists x',y',t.\ R^2_{\ell_2}(x',y') \wedge t \geq 0 \wedge x = f^x_{\ell_2}(x',y')(t) \wedge y = f^y_{\ell_2}(x',y')(t) \wedge Inv_{\ell_2}(x,y) \\
& = & \exists x',y',t.\ x' \geq 0 \wedge \underbrace{y' = 0}_{elim.\ y'} \wedge x' \leq y' + 1 \wedge y' \leq x' + 1 \wedge t \geq 0 \wedge \\
& & \underbrace{x = x'}_{elim.\ x'} \wedge y = y' + t \wedge y \leq x + 1 \\
& = & \exists t.\ x \geq 0 \wedge x \leq 1 \wedge 0 \leq x + 1 \wedge t \geq 0 \wedge \underbrace{y = t}_{elim.\ t} \wedge y \leq x + 1 \\
& = & x \geq 0 \wedge x \leq 1 \wedge y \geq 0 \wedge y \leq x + 1 \\
\\
R^4_{\ell_1}(x,y) & = & \mathcal{D}^+(R^3_{\ell_2}) \\
& = & \exists x',y'.\ R^3_{\ell_2}(x',y') \wedge x = \mu^x_{\ell_2 \to \ell_1}(x',y') \wedge y = \mu^y_{\ell_2 \to \ell_1}(x',y') \wedge Inv_{\ell_1}(x,y) \\
& = & \exists x',y'.\ x' \geq 0 \wedge x' \leq 1 \wedge y' \geq 0 \wedge y' \leq x' + 1 \wedge \underbrace{x = x'}_{elim.\ x'} \wedge \underbrace{y = y'}_{elim.\ y'} \wedge x \leq y + 1 \\
& = & x \geq 0 \wedge x \leq 1 \wedge y \geq 0 \wedge y \leq x + 1 \wedge x \leq y + 1 \\
R^4_{\ell_2}(x,y) & = & \mathcal{D}^+(R^3_{\ell_1}) \\
& = & \exists x',y'.\ R^3_{\ell_1}(x',y') \wedge x = \mu^x_{\ell_1 \to \ell_2}(x',y') \wedge y = \mu^y_{\ell_1 \to \ell_2}(x',y') \wedge Inv_{\ell_2}(x,y) \\
& = & \mathit{false} \\
\\
R^5_{\ell_1}(x,y) & = & \mathcal{T}^+_{\ell_1}(R^4_{\ell_1}) \\
& = & \exists x',y',t.\ R^4_{\ell_1}(x',y') \wedge t \geq 0 \wedge x = f^x_{\ell_1}(x',y')(t) \wedge y = f^y_{\ell_1}(x',y')(t) \wedge Inv_{\ell_1}(x,y) \\
& = & \exists x',y',t.\ x' \geq 0 \wedge x' \leq 1 \wedge y' \geq 0 \wedge y' \leq x' + 1 \wedge x' \leq y' + 1 \wedge \\
& & t \geq 0 \wedge \underbrace{x = x' + t}_{elim.\ x'} \wedge \underbrace{y' = y}_{elim.\ y'} \wedge x \leq y + 1 \\
& = & \exists t.\ x - t \geq 0 \wedge x - t \leq 1 \wedge y \geq 0 \wedge y \leq x - t + 1 \wedge x - t \leq y + 1 \wedge \\
& & t \geq 0 \wedge x \leq y + 1 \\
& = & \exists t.\ \underbrace{x \geq t}_{upper\ bound} \wedge \underbrace{x - 1 \leq t}_{lower\ bound} \wedge y \geq 0 \wedge \underbrace{t \leq x - y + 1}_{upper\ bound} \wedge \underbrace{x - y - 1 \leq t}_{lower\ bound} \wedge \\
& & \underbrace{t \geq 0}_{lower\ bound} \wedge x \leq y + 1 \\
& = & x - 1 \leq x \wedge x - 1 \leq x - y + 1 \wedge x - y - 1 \leq x \wedge x - y - 1 \leq x - y + 1 \wedge \\
& & 0 \leq x \wedge 0 \leq x - y + 1 \wedge y \geq 0 \wedge x \leq y + 1 \\
& = & y \leq 2 \wedge 0 \leq x \wedge y \leq x + 1 \wedge y \geq 0 \wedge x \leq y + 1 \\
R^5_{\ell_2}(x,y) & = & \mathcal{T}^+_{\ell_2}(R^4_{\ell_2}) \\
& = & \exists x',y',t.\ R^4_{\ell_2}(x',y') \wedge t \geq 0 \wedge x = f^x_{\ell_2}(x',y')(t) \wedge y = f^y_{\ell_2}(x',y')(t) \wedge Inv_{\ell_2}(x,y) \\
& = & \mathit{false}
\end{array}
$$

This computation sequence will not terminate, since each new iteration reaches some new states, but none of the computed sets intersect with the sets of bad states (which are actually not reachable).

6.3 Backward Analysis

There is a similar backward approach for the fixed-point-based reachability analysis of linear hybrid automata I. Instead of starting from the initial set and computing successors like in the forward approach, the backward search starts from a target set, defined as the set of states violating the property to be proved, and computes stepwise predecessors. The algorithm terminates if it finds the least fixed-point for the reversed one-step relation, thereby determining the set of states from which the target set can be reached. If the intersection of the resulting set with the initial set is empty, the property holds, otherwise the property does not hold.

Analogously to the forward time closure for the time steps and the postcondition for discrete steps in the forward approach, we define for the reversed steps a backward time closure for time steps and a precondition for discrete steps.

Definition 6.6. *We define the* backward time closure $\langle P\rangle_l^{\swarrow}$ *of* $P \subseteq V$ *at* $l \in Loc$ *as the set of valuations from which it is possible to reach a valuation in* P *by letting time progress:* $\langle P\rangle_l^{\swarrow}$

$$\nu' \in \langle P\rangle_l^{\swarrow} \quad \textit{iff} \quad \exists \nu \in P.\ \exists t \in \mathbb{R}_{\geq 0}.\ tcp_l[\nu'](t) \wedge \nu = f_l[\nu'](t).$$

We extend the definition to regions $R = \cup_{l \in Loc}(l, R_l)$ *as follows:* $\langle R\rangle^{\swarrow}$

$$\langle R\rangle^{\swarrow} = \cup_{l \in Loc}(l, \langle R_l\rangle_l^{\swarrow}).$$

We define the precondition $pre_e[P]$ *of* P *with respect to an edge* $e = (l, a, \mu, l')$ *as the set of valuations from which it is possible to reach a valuation from* P *by* e*:* $pre_e[P]$

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

For regions $R = \cup_{l \in Loc}(l, R_l)$ *we define* $pre[R]$

$$pre[R] = \cup_{e=(l',a,\mu,l) \in Edge}(l', pre_e[R_l]).$$

Note that, due to the τ-transitions, $R \subseteq pre[R]$. Similarly, due to time steps of duration 0 we have $R \subseteq \langle R\rangle^{\swarrow}$.

Lemma 6.3. *For all linear hybrid automata I, if* $P \subseteq V$ *is a linear set of valuations, then for all* $l \in Loc$ *and* $e \in Edge$*, both* $\langle P\rangle_l^{\swarrow}$ *and* $pre_e[P]$ *are linear sets of valuations.*

For a target state set we define its initial region as the set of states from which the target set is reachable.

Definition 6.7. *Given a region* $R \subseteq \Sigma$*, the* initial region $(\mapsto^* R) \subseteq \Sigma$ *of* R *is the set of all states from which a state in* R *is reachable:* $(\mapsto^* R)$

$$\sigma \in (\mapsto^* R) \quad \textit{iff} \quad \exists \sigma' \in R.\ \sigma \rightarrow^* \sigma'.$$

The following lemma states that if the backward algorithm terminates, it determines the states from which the target region is reachable.

Lemma 6.4. *Let* $R = \cup_{l \in Loc}(l, R_l)$ *be a region of the linear hybrid automaton I A. The initial region* $(\mapsto^* R) = \cup_{l \in Loc}(l, I_l)$ *of* R *is the least fixed-point of the equation*

$$X = \langle R \cup pre[X]\rangle^{\swarrow}$$

or, equivalently, for all locations $l \in Loc$, the set I_l of valuations is the least fixed-point of the set of equations

$$X_l = \langle R_l \cup \bigcup_{e=(l,a,\mu,l') \in Edge} pre_e[X_{l'}] \rangle_l^{\swarrow}.$$

Example 6.2 (Example backward reachability computation). *Consider again the example automaton from Figure ?? and the same representations of intial sets, activities, invariants, transition relations and bad states as in Example 6.1.*

The backward reachability computation generates the following set representations:

$$\begin{array}{lcl}
R^0_{\ell_1}(x,y) & = & Bad_{\ell_1}(x,y) \wedge Inv_{\ell_1}(x,y) \quad = \quad x = y+2 \wedge x \leq y+1 \quad = \quad \mathit{false} \\
R^0_{\ell_2}(x,y) & = & Bad_{\ell_2}(x,y) \wedge Inv_{\ell_2}(x,y) \quad = \quad x = y+2 \wedge y \leq x+1 \\
\\
R^1_{\ell_1}(x,y) & = & \mathcal{T}^-_{\ell_1}(R^0_{\ell_1}) \\
& = & \exists x', y', t.\ R^0_{\ell_1}(x',y') \wedge t \geq 0 \wedge x' = f^x_{\ell_1}(x,y)(t) \wedge y' = f^y_{\ell_1}(x,y)(t) \wedge Inv_{\ell_1}(x,y) \\
& = & \mathit{false} \\
R^1_{\ell_2}(x,y) & = & \mathcal{T}^-_{\ell_2}(R^0_{\ell_2}) \\
& = & \exists x', y', t.\ R^0_{\ell_2}(x',y') \wedge t \geq 0 \wedge x' = f^x_{\ell_2}(x,y)(t) \wedge y' = f^y_{\ell_2}(x,y)(t) \wedge Inv_{\ell_2}(x,y) \\
& = & \exists x', y', t.\ x' = y' + 2 \wedge t \geq 0 \wedge \underbrace{x' = x + t}_{\mathit{elim.}\ x'} \wedge \underbrace{y' = y}_{\mathit{elim.}\ y'} \wedge x \leq y + 1 \\
& = & \exists t.\ \underbrace{x + t = y + 2}_{\mathit{elim.}\ t} \wedge t \geq 0 \wedge x + t \leq y + 1 \\
& = & y - x + 2 \geq 0 \wedge x + y - x + 2 \leq y + 1 \\
\\
& = & \mathit{false}
\end{array}$$

Already the first iteration does not yield any new state, i.e., the algorithm terminates. Since none of the computed stes intersects with the initial state sets, no bad states can be reached from any initial state. (Note that it is even not possible to reach a bad state from any good state.)

6.4 Approximative Analysis

If the (forward or backward) iterative techniques does not converge, we can compute *over-approximations* of the sets

- $(I \mapsto^*)$ of states which are reachable from the initial states I (forward analysis), or
- $(\mapsto^* R)$ of states from which the region R is reachable (backward analysis).

Below we discuss two approaches for over-approximation: the first one is based on building convex hulls, and the second one is a widening technique.

1. Instead of computing the *union* of sets, we can compute their *convex hull*, i.e., the smallest convex polyhedron containing the operands of the union (see Figure 6.4). Though this set over-approximates the exact result, it may help the algorithms to terminate. On the one hand, if with the over-approximation we can show the correctness of the property we want to prove, then we are happy with the result: if the property holds even for the over-approximation then if holds also for the over-approximated reachable set. On the other hand, if the proof fails, then, due to the over-approximation, it does not mean that the property does not hold: those states of the over-approximation that violate the property

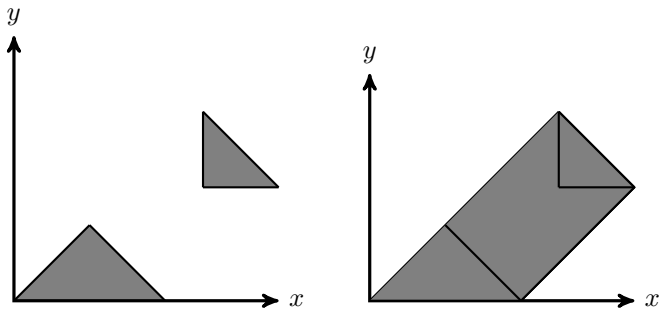


Figure 6.4: Two sets (left) and their convex hull (right)

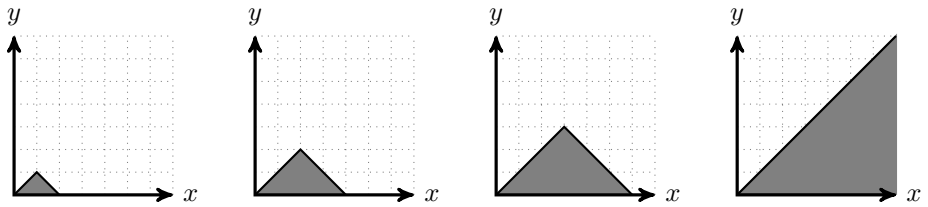


Figure 6.5: A sequence of three sets (left three pictures) and their widening (right)

may lie outside of the exact, over-approximated set and are thus perhaps not reachable. In this case we must try to find a more accurate over-approximation.

2. To enforce the convergence of iterations, we can apply a *widening technique*. The basic idea is to extrapolate the limit of the state set sequence occurring in the non-terminating fixed-point computation. The standard widening algorithm applies the widening for at least one location in each *loop* of the hybrid automaton graph. Figure 6.5 illustrates the widening technique.

6.5 Minimization

In this section we discuss another approach called *minimization* for the analysis of linear hybrid automata I, based on abstraction and abstraction refinement. We introduce a forward method but it is also possible to define it for a backward search.

Assume a linear hybrid automaton I and a safety property whose validity we want to check. The property divides the state space of the hybrid automaton into a set of "good" states that satisfy the property and a set of "bad" states that violate it. Let R_{bad} denote the set of violating states. To check the validity of the property we check if a state from R_{bad} is reachable.

The abstraction is based on partitioning the state space of a linear hybrid automaton into a finite set $\Pi = \{R_{bad}, R_1, \ldots, R_n\}$ of regions with $R_{bad} \cap R_i = \emptyset$ for all $1 \leq i \leq n$, $R_i \cap R_j = \emptyset$ for all $1 \leq i < j \leq n$, and $\Sigma = R_{bad} \cup \bigcup_{i=1}^{n} R_i$. Each such partitioning induces a LSTS being an abstraction of the linear hybrid automaton I. The abstract states of the LSTS are the regions of the partitioning. The regions containing at least one concrete initial state are the abstract initial states. There is a transition from a region R to a region R' of the partitioning, denoted by $R \mapsto R'$, iff from at least one state in R at least one state in R' is reachable in one step. Since

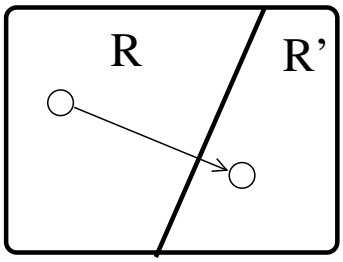


Figure 6.6: The next relation $\mapsto$ on regions

we are only interested in the reachability of bad states, we define no successors for R_{bad}. The abstract transitions are formalized as follows:

$\mapsto$

Definition 6.8. *The next relation $\mapsto$ on regions is defined by*

$$R \mapsto R' \quad \text{iff} \quad R \neq R_{bad} \ \wedge \ \exists \sigma \in R.\ \exists \sigma' \in R'.\ \sigma \to \sigma'.$$

Figure 6.6 illustrates the next relation.

Such an abstraction in general over-approximates the behaviour of the concrete system: For each reachable state of the concrete system the region of the abstraction that contains that state is also reachable. However, there may be regions reachable in the abstraction that contain no states reachable in the concrete system.

That implies on the one hand, that if R_{bad} is not reachable in the abstraction then the property holds for the concrete system. But on the other hand, from the reachability of R_{bad} in the abstraction we cannot conclude that the property does not hold for the original system. However, we can define a sufficient condition under that the second implication also holds, i.e., a condition that assures that R_{bad} is reachable in the abstraction if and only if the concrete system violates the property. This condition is that all regions reachable in the abstraction have at least one state reachable in the concrete system. The minimization algorithm starts with an initial partitioning and splits regions of the partitioning iteratively until it satisfies that sufficient condition. Note that, since the reachability problem for linear hybrid automata I is not decidable, the refinement loop does not always terminate. But in case it terminates, the abstraction is finite, and we can answer the reachability question.

How can we be sure that a region R reachable in the abstraction contains at least one state reachable in the concrete system? First we only know that all initial regions contain at least one initial state by definition. Now assume a reachable region R that contains at least one state $\sigma \in R$ reachable in the concrete system, and assume a successor region R' of R with $R \mapsto R'$. From $R \mapsto R'$ we conclude that there is a state in R with a successor state in R', however, we do not know if this state is σ. But, if *all* states in R have a successor state in R', then also σ has a successor state $\sigma' \in R'$, and from the reachability of σ together with $\sigma \to \sigma'$ we can conclude that there is at least one reachable state in R'.

Definition 6.9. *Let Π be a partitioning of the state space Σ and let $R, R' \in \Pi$. The region R is called* stable for R' *iff*

$$R \mapsto R' \quad \text{implies} \quad \forall \sigma \in R.\ \{\sigma\} \mapsto R'.$$

We call R stable iff it is stable for all regions in Π. We call Π stable iff all reachable regions of Π are stable.

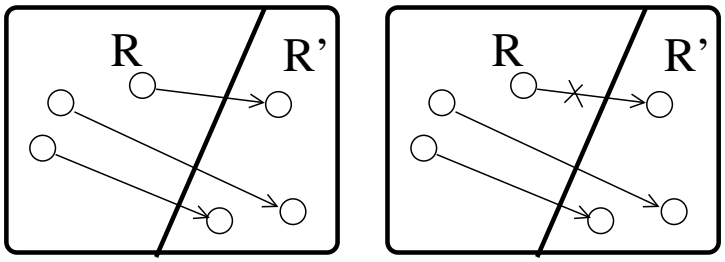


Figure 6.7: Stability of regions: a stable region (left) and a non-stable one (right)

Figure 6.7 illustrates the stability of regions.

Now we come to the algorithm as specified by Figure 6.8. The set of initial states of the concrete system is denoted by I, and R_{bad} is the set of "bad" states. The algorithm stores the current partitioning in Π. Initially there are two regions in the patitioning: the region R_{bad} contains all "bad" states and the region $\Sigma \backslash R_{bad}$ the "good" states.

The algorithm uses two sets *reach* and *completed*. In the set *reach* we store those reachable regions of the current partitioning for which we know that they contain at least one concrete state that is reachable in the concrete system. In the set *completed* $\subseteq$ *reach* we store regions from which we know that their successor regions are all in *reach*, i.e., regions that currently cannot be used to derive further in the concrete system reachable regions. Initially, *reach* contains those regions of the initial partitioning that contain at least one concrete intial state. The set *completed* is initially empty.

In each refinement step we determine a reachable region $R \in$ *reach* from that we already know that it has at least one reachable state, but we do not yet know if all of its successor regions contain reachable states, i.e., such that R is not in *completed*. For all those successor regions of R for which R is stable we can conclude that also they contain at least one reachable state, thus we put them into the *reach* set.

If, after that update, all successor regions of R are in *reach*, i.e., they all have at least one reachable state, then we put R into the *completed* set.

Otherwise, if there is still a successor region $R' \notin$ *reach* of R then R is not stable for R'. We use such an R', found at last, to split R into two parts, one containing all states with a successor in R' and a second part containing the rest. The splitting of a region is formalized by the following definition:

Definition 6.10. $\mathsf{split}(\Pi, R, R')$

$$\mathsf{split}(\Pi, R, R') \quad := \quad \begin{cases} \{R'', R \setminus R''\} & \text{if } R'' = \mathsf{pre}[< R' >^{\checkmark}] \cap R \wedge R'' \neq \emptyset \wedge R'' \neq R, \\ \{R\} & \text{otherwise.} \end{cases}$$

Figure 6.9 illustrates the splitting mechanism.

We split R according to the splitting result remembered in $S = \{S_1, S_2\}$, and update the partitioning. The *reach* set gets updated in that we remove R and add S_i, $i = 1, 2$, if they contain concrete initial states. Note that, though we know that there is a concrete state either in S_1 or in S_2 that is reachable in the concrete system, we do not know which of both sets contains it. Thus we can add S_1 or S_2 to *reach* only if they contain concrete initial states. Note also that all other elements $R' \neq R$ in *reach* can stay in the set. Previous predecessors of R are now predecessors of S_1 and/or S_2. For such predecessors that are in *completed* we check if still all of their successors are in *reach*, and remove them from *completed* if it is not the case. All other regions in *completed* remain in the set.

```
minimize(Σ, R_bad) {
  Π := {R_bad, Σ \ R_bad}; reach := {R ∈ Π | R ∩ I ≠ ∅}; completed := ∅;
  while (R_bad ∉ reach ∧ reach ≠ completed) {
    choose R ∈ (reach \ completed); S := ∅;
    for each (R' ∈ (Π \ reach) with R ↦ R') {
      reach' := split(Π, R, R');
      if (reach' = R) then reach := reach ∪ {R'};
      else S := reach';
    }
    if (S = ∅) then completed := completed ∪ {R};
    else {
      Π := (Π \ {R}) ∪ S;
      reach := (reach \ {R}) ∪ {S_i ∈ S | S_i ∩ I ≠ ∅};
      completed := completed \ {R' ∈ Π | ∃S_i ∈ (S\reach). R' ↦ S_i};
    }
  }
  return R_bad ∈ reach;
}
```

Figure 6.8: The minimization algorithm

We observe that, since "bad" regions do not have outgoing transitions in the abstract LSTS, they are never split. Thus there is always a single "bad" region in the partitioning.

Before each iteration we check if one of the termination conditions hold: If $R_{bad} \in \mathit{reach}$ then the system violates the property. Otherwise, if $R_{bad} \notin \mathit{reach}$ but $\mathit{reach} = \mathit{completed}$ then R_{bad} is not reachable in the abstraction, and the property holds.

Note that if the regions R_{bad} and I are linear, all regions that are constructed by the procedure are linear.

Lemma 6.5. *The procedure in Figure 6.8 returns TRUE iff* $I \mapsto^* R_{bad}$.

Example 6.3. *Assume the linear hybrid automaton I shown in Figure 6.10. We want to prove that* $0 \leq y$ *always holds.*

We have

$$\begin{aligned} R_{bad} &= (\ell_1, y < 0) \cup (\ell_2, 0 \leq x \wedge y < 0) \\ R_1 &= (\ell_1, 0 \leq y) \cup (\ell_2, 0 \leq x \wedge 0 \leq y) \end{aligned}$$

The algorithm initializes

$$\begin{aligned} \Pi &= \{R_{bad}, R_1\} \\ \mathit{reach} &= \{R_1\} \\ \mathit{completed} &= \emptyset\ . \end{aligned}$$

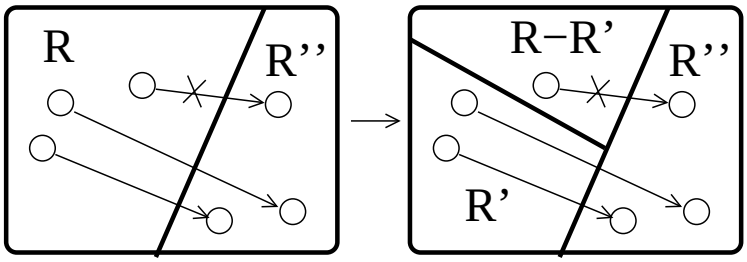


Figure 6.9: The splitting of regions

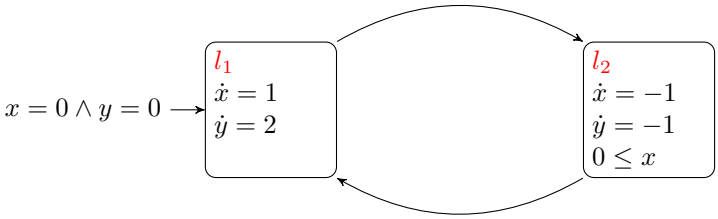


Figure 6.10: Leaking gas burner

Since $R_{bad} \notin reach$ *and* $reach \neq completed$ *the main loop is entered. We choose the only element* $R_1 \in reach$*. Its only successor region is* R_{bad}*. We first compute the time predecessor of* R_{bad}*:*

$$\begin{aligned} \langle R_{bad} \rangle^{\swarrow} &= \langle (\ell_1, y < 0) \cup (\ell_2, 0 \leq x \wedge y < 0) \rangle^{\swarrow} \\ &= \langle (\ell_1, y < 0) \rangle^{\swarrow} \cup \langle (\ell_2, 0 \leq x \wedge y < 0) \rangle^{\swarrow} \end{aligned}$$

To compute $\langle (\ell_1, y < 0) \rangle^{\swarrow}$ *assume a time step resulting in a state from* $(\ell_1, y < 0)$*. Then the control is in* ℓ_1 *also before the time step. For the valuation, if* x *and* y *denote the values before the time step, then after the time step the values change to* $x + t$ *and* $y + 2t$ *for some* $0 \leq t$*, and we know that* $y + 2t < 0$*. We have to eliminate* t *from the equation set*

$$0 \leq t \wedge y + 2t < 0,$$

i.e.,

$$0 \leq t \wedge t < -y/2$$

which yield $\langle (\ell_1, y < 0) \rangle^{\swarrow} = (\ell_1, y < 0)$*.*

To compute $\langle (\ell_2, 0 \leq x \wedge y < 0) \rangle^{\swarrow}$ *assume a time step resulting in a state from* $(\ell_2, 0 \leq x \wedge y < 0)$*. Then before the time step control is in* ℓ_2*. Let again* x *and* y *denote the variable values before the time step. The time step changes the values to* $x - t$ *and* $y - t$ *for some* $0 \leq t$*. Due to the invariant* $0 \leq x$ *and* $0 \leq x - t$*, and since the target state should be from* R_{bad} *we have* $y - t < 0$*. Eliminating* t *from the equation system*

$$0 \leq x \wedge 0 \leq x - t \wedge 0 \leq t \wedge y - t < 0,$$

i.e.,

$$0 \leq x \wedge t \leq x \wedge 0 \leq t \wedge y < t,$$

we get $0 \leq x \wedge y < x$. *Thus* $\langle(\ell_2, 0 \leq x \wedge y < 0)\rangle^{\swarrow} = (\ell_2, 0 \leq x \wedge y < x)$.

Collecting the above information, $\langle R_{bad}\rangle^{\swarrow} = (\ell_1, y < 0) \cup (\ell_2, 0 \leq x \wedge y < x)$.

Now we compute the discrete step predecessor of this set.

$$\begin{array}{l} pre\,[\langle R_{bad}\rangle^{\swarrow}] = \\ pre\,[(\ell_1, y < 0) \cup (\ell_2, 0 \leq x \wedge y < x)] = \\ \underbrace{(\ell_1, y < 0)}_{\tau_{\ell_1}} \cup \underbrace{(\ell_2, 0 \leq x \wedge y < x)}_{\tau_{\ell_2}} \cup \underbrace{(\ell_2, 0 \leq x \wedge y < 0)}_{edge\ from\ \ell_2\ to\ \ell_1} \cup \underbrace{(\ell_1, 0 \leq x \wedge y < x)}_{edge\ from\ \ell_1\ to\ \ell_2} = \\ (\ell_1, y < 0 \vee 0 \leq y < x) \cup (\ell_2, (0 \leq x \wedge y < 0) \vee (0 \leq y < x)) \end{array}$$

The intersection of this predecessor set with R_1 *yields*

$$\begin{array}{lcl} pre\,[\langle R_{bad}\rangle^{\swarrow}] \cap R_1 & = & [(\ell_1, y < 0 \vee 0 \leq y < x) \cup (\ell_2, (0 \leq x \wedge y < 0) \vee (0 \leq y < x))] \cap \\ & & [(\ell_1, 0 \leq y) \cup (\ell_2, 0 \leq x \wedge 0 \leq y)] \\ & = & (\ell_1, 0 \leq y < x) \cup (\ell_2, 0 \leq y < x) \\ & =: & R_2. \end{array}$$

We define

$$R_3 := R_1 \backslash R_2 = (\ell_1, 0 \leq x \leq y) \cup (\ell_2, 0 \leq x \wedge 0 \leq y \wedge x \leq y).$$

Thus we have $split(\Pi, R_1, R_{bad}) = \{R_2, R_3\}$. *The corresponding updates result in*

$$\begin{array}{rcl} \Pi & = & \{R_{bad}, R_2, R_3\} \\ reach & = & \{R_3\} \\ completed & = & \emptyset \end{array}$$

In the next iteration the termination conditions are still not met thus we execute the loop once more. For $R_3 \in reach$ *we have no successor regions, thus the region does not get split and the update results in*

$$\begin{array}{rcl} \Pi & = & \{R_{bad}, R_2, R_3\} \\ reach & = & \{R_3\} \\ completed & = & \{R_3\}. \end{array}$$

In the next iteration we detect that the termination condition $reach = completed$ *holds.*

Since $R_{bad} \notin reach$, *the algorithm returns that the property holds.*

The minimization of linear hybrid automata I is a special case of a more general approach frequently used for the reachability analysis of general hybrid systems. The general approach defines an initial partitioning of the state space and refines it by region splitting until it becomes fine enough to prove or violate the requested safety property. The different instances of this general approach use different methods to determine the regions to be split and the splitting itself.

Chapter 7

Linear Hybrid Automata II

In the previous chapter we have seen an approach for the reachability analysis of hybrid systems with linear behavior (i.e., where the derivatives are independent of the current state) based on fixed point computations. There we represented the states sets logically by formulas.

In this chapter we discuss state set representation and reachability analysis techniques for a more general class of hybrid automata, where the dynamics (flows) of the systems are given by linear differential equations. Due to the vast variety of hybrid systems reachability analysis algorithms and tools, we focus on *flowpipe-construction-based* techniques.

7.1 Flowpipe-construction-based Reachability Analysis

At the heart of hybrid systems safety verification is the computation of the set of reachable states. Basic safety properties specify a set of *bad* states, whose reachability is safety-critical. Whenever the intersection of the reachable state set with the bad state set is empty the system can be considered as safe.

As the reachability problem for hybrid systems is in general undecidable, most approaches aim at computing an *over-approximation* of the set of states reachable via bounded flow durations and a bounded number of jumps. The bound on the flow duration is usually called the *time horizon*, the bound on the number of jumps the *jump depth*. Due to the over-approximation, results of such analysis methods allow to declare a system safe (within the specified execution bounds) whenever the intersection of the bad state set with the obtained reachable state set is empty, but provide no conclusion when this intersection is not empty.

Current reachability analysis tools for hybrid systems use different approaches to solve this problem. Some approaches are based on *theorem proving* and combine deductive, real algebraic, and computer-algebraic prover technologies. Such techniques can prove also unbounded safety, but in most cases need user interaction. Other *SMT-solving-based approaches* use logical characterisations of the reachability problem and employ bounded model checking and *SAT-module-theories* (SMT) solving technologies to check safety.

Here we focus on *flowpipe-construction-based* approaches for reachability analysis. These approaches iteratively compute successors of a given initial state set; to over-approximate time-evolution trajectories (*flowpipes*) for a given time horizon, they pave the flowpipe with state sets (see Figure 7.1).

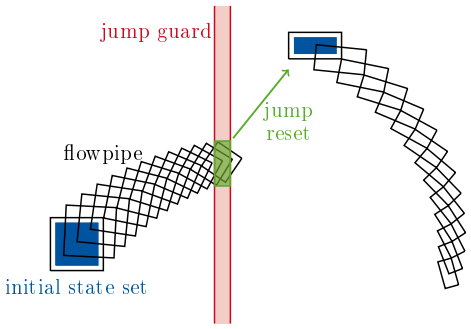


Figure 7.1: Flowpipe-construction-based reachability analysis.

7.1.1 General Reachability Analysis Algorithm

Next we describe the general algorithm for flowpipe-construction-based reachability analysis. The algorithms gets as input a hybrid system model (here we assume a hybrid automaton) and iteratively computes successors of the hybrid system's initial state set(s). The successor computation considers flows and jumps in an alternating manner. Optionally, a set of bad states could be considered as input; in this case the algorithm would terminate whenever any discovered reachable state is included in the set of bad states.

The general algorithm is presented in Algorithm 1. It maintains two sets which we assume to be globally accessible: R, a set of state sets currently known to be reachable (up to over-approximation), and R_{new}, a set of state sets which need to be processed. After initialising R_{new} with the initial state sets of the hybrid system, the algorithm picks an unprocessed state set *stateset* from R_{new} (Line 4) and computes a set of state sets whose union covers all states reachable from *stateset* via a single flow and a single jump (Line 6).

Additionally, the algorithms might put effort into finding *fixed-points* during analysis, e.g., to detect when successor state sets are fully included in previously discovered state sets (whose successors were or will already be handled by the algorithm). Such fixed-point analysis might be expensive due to frequent intersection computations, but it might pay off when leading to earlier termination of the search.

In case we are interested not only in computing the reachable state set but also in checking the reachability of a set of *bad states*, the intersection of each reachable state set with the bad state set needs to be checked for emptiness.

The flowpipe is computed by the *computeFlowPipe* method. This method returns a set R' of state sets which cover the whole flowpipe (for time-bounded search up to a given time horizon), according to the specified dynamics and the invariant in the respective location. For time-unbounded analysis, if further time evolution beyond some currently considered time horizon should be considered later, the state set covering the last flowpipe segment is added to R_{new}. Furthermore, fixed-point checks might lead to the removal of some of the state sets, and clustering and aggregation (see pages 87 and Section 7.1.5 on page 87) can be applied to replace a subset of R' by a single over-approximating state set.

After the computation of a flowpipe, all possible jump successors from all state sets in R' are computed by the method *computeJumpSuccessors* (Line 7). This computation involves the intersection of those sets with the guards of jumps starting in the respective location, and in case of a non-empty intersection, the application of the reset function to compute the successor sets (see also Figure 7.1). The resulting successor are collected, and after optional fixed-point

detection, clustering and aggregation, they are added to R and R_{new}.

This procedure terminates if R_{new}, the set of state sets which have to be processed, is empty, i.e. a fixed-point is reached. In this case, the union of all sets in R is an over-approximation of the set of reachable states for the given model. For bounded search, local time horizons can bound the length of the computed flowpipes, a predefined jump depth can bound the number of jumps along considered paths, or an iteration bound can bound the number of loop iterations. Such bounds guarantee the termination of the algorithm.

Input: Hybrid system model H.
Output: Set of reachable states R_H.
1 $R := Init_H$;
2 $R_{new} := R$;
3 **while** $R_{new} \neq \emptyset \wedge \neg termination\_cond$ **do**
4 | *let* $stateset \in R_{new}$;
5 | $R_{new} := R_{new} \backslash \{stateset\}$; /* modifies R and R_{new} */
6 | $R' := computeFlowPipe(stateset)$; /* modifies R and R_{new} */
7 | $computeJumpSuccessors(R')$;
8 **end**
9 **return** R
Algorithm 1: Abstract flowpipe-construction-based reachability analysis algorithm for hybrid systems.

7.1.2 Flowpipe Construction

One of the main ingredients of the previously presented algorithm is the computation of flowpipes: given an initial set, we need to compute an over-approximation of the time successors of this initial set within a predefined time horizon.

Flowpipe-construction-based methods discretise the time horizon into equal segments (time steps) of length δ, and iteratively compute the reachable set of states for each δ. Most algorithms over-approximate the first segment (for the time interval $[0, \delta]$) by a chosen state set representation and compute all other segments by a recurrence relation, which depends on the constant time-step size δ, the initial set in which the evolution starts, and the dynamics in the given location.

In the following we describe how to construct the flowpipe. For this computation we have to take the shape of the dynamics of the location into account and we differentiate between linear and non-linear models.

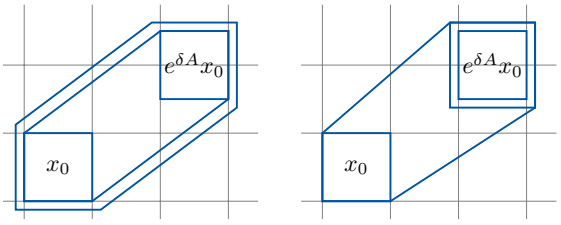
Flowpipe Construction for Linear Hybrid Automata I

The flow inside a location of an autonomous linear hybrid system is described by a system of linear ODEs

$$\dot{x}(t) = A \cdot x(t)\,. \tag{7.1}$$

A non-autonomous system is an extension of the dynamics by a time-dependent function $u(t)$, which is used to represent external inputs influencing the system evolution and results in a dynamics of the form

$$\dot{x} = A \cdot x + B \cdot u(t)\,.$$



(a) Computation of the first segment using uniform bloating.

(b) Computation of the first segment using improved bloating.

Figure 7.2: Two approaches towards the computation of the first flowpipe segment.

A solution to 7.1 has the form $x(t) = e^{tA}x_0$ specifying the state reachable from the initial state x_0 with the flow matrix A at time point t. As previously mentioned, the set of reachable states is obtained by time discretisation. For autonomous systems, computing the set of states reachable from a state set Ω_i at time δ can be obtained by the recurrence relation

$$\Omega_{i+1} = e^{\delta A}\Omega_i \,. \tag{7.2}$$

This allows to iteratively compute the set of reachable states at certain time points $i \cdot \delta, i \geq 0$ by applying a linear transformation on the previously computed segments. For non-autonomous systems an additional bloating is needed (via Minkowski sum, see below) in each iteration.

The behaviour in between these discrete time points, i.e., at time points t with $i \cdot \delta < t < (i+1) \cdot \delta$, is not yet covered by this relation. To be able to make statements about the reachable states for a time interval $[i \cdot \delta, (i+1) \cdot \delta]$, an over-approximation has to be computed (see Figure 7.2). Note that we only have to compute the first set Ω_0 over-approximating the set of states reachable from the initial state set X_0 within the time interval $[0, \delta]$, and using Equation 7.2 we can obtain the subsequent sets by applying linear transformations to Ω_0. There are two main approaches to compute the first segment, which both utilize bloating to completely cover the flowpipes.

Uniform bloating [Gir05] covers the dynamics of the autonomous part by adding a bloating factor α to the convex hull of the union of the initial set X_0 and the set $e^{\delta A}X_0$ reachable from X_0 at time δ. For non-autonomous systems, an additional factor β can be computed, which covers the influences of the external input. Using this method, the first time interval can be computed as

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

where $\mathcal{B}_{(\alpha+\beta)}$ is a ball of radius $(\alpha + \beta)$, which is added to the convex hull of the union of both sets. The radius α is dependent on the shape of the flow, the time step size and the initial set; details on its computation can be found in [Gir05]. The Minkowski sum of a ball and the convex hull of the union results in a uniform bloating in all directions (see Figure 7.2(a)). In case the modelled system is non-autonomous, the bloating is increased to reflect the influence of the external input [LG09].

Improved bloating (see Figure 7.2(b)) was first presented in [Gir04]. In contrast to uniform bloating, which computes the convex hull of the union and bloats afterwards, this approach

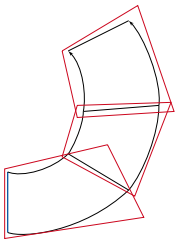


Figure 7.3: Flowpipe computation as an over-approximation of the reachable set.

bloats the reachable set at time δ and computes the convex hull of the union of the initial set and the bloated set at time δ. Thus, the first time interval can be computed as

$$\Omega_0 = conv(X_0 \cup (e^{\delta A} X_0 \oplus \mathcal{B}_{(\alpha'+\beta')})) \,, \tag{7.3}$$

where $\mathcal{B}_{(\alpha'+\beta')}$ is a ball of radius $(\alpha' + \beta')$. As for uniform bloating, the parameter α' is used to cover the autonomous dynamics inside the location and is dependent on the time step size, the flow and the initial set X_0. The additional factor β' is used to encapsulate effects of the external input. As a result, flowpipes created with this method usually are more smooth and might produce a smaller over-approximation than uniform bloating.

After computing an over-approximation of the first flowpipe segment covering reachability within the time interval $[0, \delta]$, all further segments up to the time horizon T can be obtained by iterative linear transformation applied on the previously computed segment; in case the system is non-autonomous, an additional bloating is needed to cover the influence of the external input (see Figure 7.3). Basically, this results in an alternating application of linear transformation and Minkowski sum, which in general increases the complexity of the set representation (see Section 7.1.4). In [LG09] another approach has been proposed, which allows to separate these operations to reduce the increase in the representation size.

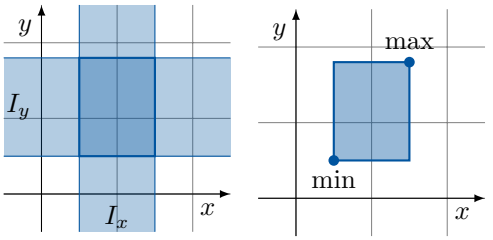
Flowpipe Construction for Non-linear Hybrid Automata

The flowpipe construction for non-linear hybrid systems is more involved, but its basic ideas are similar to those of the linear case. Also here, the flowpipe is over-approximated segment-wise, however, the over-approximation of the flowpipe segments works differently. These computations do not involve linear transformation and bloating, but employ for example Taylor models, which will be described on page 84.

Jump Successor Computation

After the flowpipe segments are over-approximated by a sequence of state sets, we need to check for each of these sets whether they intersect with the guards of outgoing transitions. For each intersecting set, we need to apply a transformation to reflect the effect of the jump's reset function. The involved computations depend on the state set representations used, and on the form of the guard condition and the reset function. E.g., a projection operation can be applied to compute the effect of setting a variable to a constant value.

After the intersection and the transformation, the resulting sets are to be processed in the target location for flowpipe construction in future iterations.



(a) A box represented as a vector of intervals.

(b) A box represented by two points.

Figure 7.4: Box representation.

7.1.3 State Set Representations

To implement the reachability analysis Algorithm 1, we need datatypes for the representation of state sets, and certain (over-approximative) operations (union, intersection, linear transformation, Minkowski sum etc.) on them. Most flowpipe-construction-based reachability analysis approaches rely on geometric representations (e.g., boxes/hyperrectangles, oriented rectangular hulls, convex polyhedra, template polyhedra, orthogonal polyhedra, zonotopes, ellipsoids) or other symbolic representations (e.g., support functions or Taylor models) for state sets.

The variety of representations is rooted in the general problem of deciding between computational effort and precision (see Section **??**). Generally, faster computations often come at the cost of precision loss and vice versa, more precise computations need higher computational effort. The representations might differ in their size, i.e., the required memory consumption, which has a further influence on the computational costs for operations on these representations. While some representations are able to perform certain operations very efficiently, other operations on the same representation, which are also needed for the analysis, can be computationally expensive.

In the following we describe some of the most popular state set representations. Let $\mathbb{I}$ be the set of all *intervals* $[a,b],[a,\infty),(-\infty,b],(-\infty,+\infty) \subseteq \mathbb{R}$ with $a,b \in \mathbb{R}$; for simplicity, we call cross-products of intervals from $\mathbb{I}$ also intervals. An interval is *bounded* if both of its bounds are finite.

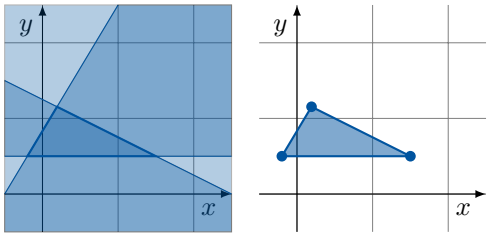
Boxes A box is defined by the cross product of intervals, one for each dimension of the state space (see Figure 7.4(a)).

Definition 7.1 (Box). *A set $\mathcal{B} \subseteq \mathbb{R}^d$ is a* box *if there exist intervals $I_1,\ldots,I_d \in \mathbb{I}$ such that*

$$\mathcal{B} = I_0 \times \ldots \times I_d\,.$$

A box can be represented by the sequence $(I_1,\ldots,I_d)$ of the intervals defining it. Alternatively, we can represent a box by its minimal and its maximal point (see Figure 7.4(b)). Boxes are well-suited for fast computations in flowpipe construction, however, they often lead to large over-approximations. Boxes are widely used also in other fields such as in *interval constraint propagation (ICP)*, which itself is used for SMT-solving-based reachability analysis of hybrid systems [FHR+07]. Implementations of boxes are also contained in most polytope libraries.

Convex polytopes A convex polyhedron can be defined by the intersection of finitely many halfspaces.



(a) A $\mathcal{H}$-representation of a convex polytope. (b) A $\mathcal{V}$-representation of a convex polytope.

Figure 7.5: Convex polytope representation.

Definition 7.2 (Convex polyhedron). *A set* $\mathcal{P} \subseteq \mathbb{R}^d$ *is a* convex polyhedron *if there are* $n \in \mathbb{N}$ *and* $c_i \in \mathbb{R}^n$, $d_i \in \mathbb{R}$, $i = 1, \ldots, n$ *such that*

$$\mathcal{P} = \bigcap_{i=1}^{n} h_i \text{ where } h_i = \{x \in \mathbb{R}^d \mid c_i x \leq d_i\}.$$

In the following we restrict ourselves to closed convex polyhedra called *convex polytopes*, which have two widely used representations. An *$\mathcal{H}$-representation* (C, d) consists of a $n \times d$ matrix C with c_i being its ith row and an n-dimensional vector d with d_i being its ith components, and specifies the polytope $\mathcal{P} = \bigcap_{i=1}^{n} \{x \mid c_i x \leq d_i\}$ (see Figure 7.5(a)). Alternatively, a $\mathcal{V}$-representation consists of a finite set V of d-dimensional points and specifies a polytope as the convex hull $\mathcal{P} = conv(V)$ of those points (see Figure 7.5(b)).

Polytopes are a more complex representation as for instance boxes but allow for a more precise description of a set. The two presented representations are complementary in the complexity of the required operations for reachability analysis. Computing the convex hull of union requires little computational effort in the $\mathcal{V}$-representation but is hard in the $\mathcal{V}$-representation, whereas intersection can easily be performed with the $\mathcal{H}$-representation of a polytope but it is hard in the $\mathcal{V}$-representation. Unfortunately, conversion between the two representation requires either facet enumeration of a set of vertices or vertex enumeration of a set of hyperplanes, which are both computationally difficult. There are libraries already providing implementations of convex polytopes such as [BHZ08] [GJ00], but as they are intended to provide general purpose implementations, functionality required for hybrid systems reachability analysis is not fully optimized (e.g. PPL does currently not provide Minkowski sum implementations).

Zonotopes Zonotopes, sometimes also referred to as parallelotopes, are point-symmetric sets that can be defined as the Minkowski sum of a finite set of line segments shifted to a given centre point (see Figure 7.6).

Definition 7.3 (Zonotope). *A set* $\mathcal{Z} \subseteq \mathbb{R}$ *is a* zonotope *if there is a* center $c \in \mathbb{R}^n$ *and a finite set* $G = \{g_1, \ldots, g_n\}$ *of* generators $g_i \in \mathbb{R}^d$ *such that*

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

Zonotopes can be represented by their defining vectors $(c, g_1, \ldots, g_n)$, i.e., by their centre and the generators. Zonotopes allow for a fast computation of the operations union and Minkowski

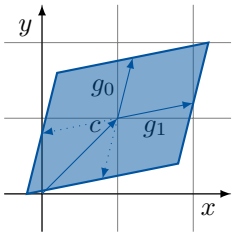


Figure 7.6: Zonotope representation of a set centred at c using two generators g_0 and g_1.

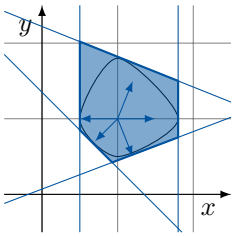


Figure 7.7: A set represented by a support function evaluated in 5 directions.

sum. However, intersections with halfspaces or other zonotopes are hard to compute. Zonotopes are often used due to their reduced storage requirement in comparison to for example convex polytopes. Zonotope implementations are contained in the C++ library polymake as well as in the Matlab tool collection CORA.

Support functions Support functions are, in contrast to the above presented representations, a symbolic representation, which allows queries for specific directions and will return a support value (see Figure 7.7).

Definition 7.4 (Support function). *A* support function *is a function* $\sigma : \mathbb{R}^d \to \mathbb{R}$ *defining a set*

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

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

The definition of support functions allows for an implementation, which reduces computation time during reachability analysis significantly. This is due to the fact that while other representations always maintain the whole object, support functions only need to store the operation and its parameters. Whenever the support value of a given direction is queried, the stored operations are applied reversed on the direction vector instead of applying them to the whole object. This implies that only directions are computed which are of interest instead of the whole object. The disadvantage of this representation is, that unless infinitely many directions are queried, the exact shape of the set is hidden. Support functions are used for example in the tool SPACEEX, which successfully makes use of algorithms optimized for support functions (LGG [LGG10, FGD+11], STC [FKL13, Fre15]).

Taylor models are a state set representation, which can be used for the reachability analysis of non-linear hybrid systems. The basic idea is to over-approximate the given dynamics by a

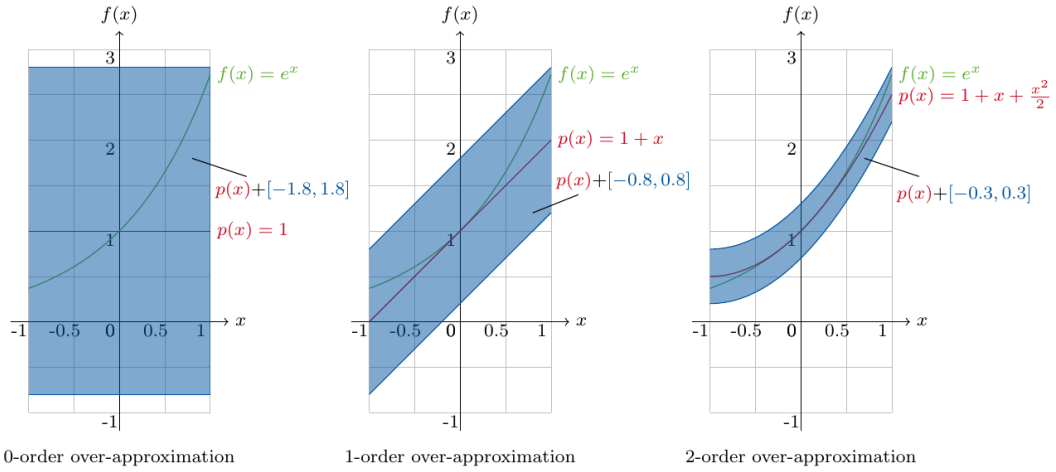


Figure 7.8: Taylor model approximations of different degrees.

polynomial of fixed degree k bloated by a suitable interval I, such that for a fixed initial set and a fixed time interval, all solutions of the ODE are contained in the area spanned by the Minkowski sum of the polynomial and the interval (see Figure 7.8).

Formally, assume a bounded domain $D \in \mathbb{I}^d$. A given polynomial p is a *k-order Taylor approximation* of a function $f : D \to \mathbb{R}$ iff

(a) all partial derivatives of f up to order k exist and are continuous, denoted by $f \in C^k$, and

(b) $f(c) = p(c)$ for the centre point c of D and for each $0 < m \leq k$, all of the order m partial derivatives of f and p coincide at c.

For any $f, g : D \to \mathbb{R}$, $f, g \in C^k$ and $k \geq 0$, we write $f \equiv_k g$ iff there is a polynomial p which is a k-order approximation of both f and g. Taylor models are based on the equivalence relation $\equiv_k$.

Definition 7.5 (Taylor Model). *A* Taylor model *of order $k > 0$ over a bounded domain $D \in \mathbb{I}^d$ is a pair (p, I) of a polynomial p of degree at most k over d variables x and a* remainder interval *$I \in \mathbb{I}$. We say that (p, I) is a* k-order over-approximation *of a function $f : D \to \mathbb{R}$, written $f \in (p, I)$, iff (i) $p \equiv_k f$ and (ii) $\forall x \in D.\, f(x) \in p(x) + I := \{p(x) + y \mid y \in I\}$.*

7.1.4 Operations on State Set Representations

Flowpipe-construction-based reachability analysis algorithms need to apply certain *operations* on sets, whose complexity depends on the state set representation used. These operations include the (convex hull of) union, intersection, Minkowski sum, linear transformation as well as tests for emptiness and membership. Assume a domain D, and subsets $A, B, S \subseteq D$.

- *conv*$(\cdot \bigcup \cdot)$ (*union*): As convex sets are not closed under the operation union, for convex state set representations the convex hull of the union is computed; nevertheless, we often refer to this operation just as union:

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

	$\cdot \bigcup \cdot$	$\cdot \bigcap \cdot$	$\cdot \oplus \cdot$	$A(\cdot)$
Box			+	
$\mathcal{H}$-polytope	-	+	-	-
$\mathcal{V}$-polytope	+	-	+	+
Zonotope			+	+
Support function	+	-	+	+

Table 7.1: State set operations and their complexity

The convex hull of the union of two sets is required for the computation of the first segment of a flowpipe and in case aggregation of segments is used.

- $\cdot \bigcap \cdot$ (*intersection*): The intersection of two sets is defined as

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

Intersection of two sets is required whenever a flowpipe segment is checked against the invariant of the current location, for checking whether a guard condition holds, for checking the reachability of bad states, and for fixed-point detection. Note that all mentioned checks also imply a test for emptiness of the result of the intersection.

- $\cdot \oplus \cdot$ (*Minkowski sum*): The Minkowski sum is the set-theoretic equivalent of addition:

$$A \oplus B = \left\{ x \mid x = a + b, a \in A, b \in B \right\} .$$

In case of an autonomous system, Minkowski sum is only required for the computation of the first flowpipe segments (bloating). In case of a non-autonomous system, additional bloating for each segment is added via the application of the Minkowski sum.

- $A(\cdot)$ (*linear transformation*): The linear transformation of a given set S is defined as

$$A(S) = \left\{ x \mid x = A \cdot y + b, \, y \in S \right\} ,$$

with A a $d \times d$-dimensional transformation matrix and $b \in \mathbb{R}^n$ specifies a translation. The application of a linear transformation does not increase the representation size. In flowpipe construction, the recurrence relation (see Section 7.1.1) allows to compute the next flowpipe segment from the current one by applying linear transformation, which makes linear transformation in general a frequently used operation.

- *Test for emptiness* is a predicate checking whether a set is empty, i.e., whether $S = \emptyset$ holds.
- *Test for membership* is a further predicate which checks whether a given value is contained in the set, i.e., whether $x \in S$ for some input value $x \in D$.

These (and possibly further) operations must be defined and implemented for all state set representations used in the reachability analysis algorithm. We do not describe the single implementations here, but emphasize that the complexity of these operations might strongly differ for different representations.

To find the right balance between efficiency and precision, the choice of the state set representation as well as different kinds of optimisations play a crucial role in flowpipe-construction-based methods. Unfortunately, there is no optimal representation for which all necessary operations can be easily computed (see Table 7.1). While support functions seem to be optimal in most operations, they usually require more storage. We also can observe that representations based on points, such as $\mathcal{V}$-polytopes or boxes, perform good on operations such as union, linear transformation or Minkowski sum. Representations that are based on constraints, such as $\mathcal{H}$-polytopes, naturally perform good on intersection computations.

Besides the complexity of the single operations, one has to keep in mind that not all operations are used in the same frequency. Based on the hybrid system model we want to analyse and on the approach we utilise for reachability analysis, the usage of operations varies. For example for linear autonomous systems, the operations Minkowski sum and union are performed only once per flowpipe, whereas a similar but non-autonomous system requires a more frequent application of the Minkowski sum. When enhancing standard algorithms for reduction techniques, the operation union is used more frequent, otherwise it is used again only once for each flowpipe computation (main loop iteration). The operation intersection is generally used on every computed flowpipe segment, but when the flowpipes are holding only a small number of segments and there are more locations, its significance for computation time might be reduced. There are also some results on reducing the frequency of applications for problematic operations, see e.g. [AK12] for avoiding intersection computations.

As the complexity of some operations is representation-dependent, to improve efficiency, most algorithms change the representation for certain computations using over-approximative transformations. Another efficiency-relevant issue is the reduction of the number of state sets for which successors need to be computed by clustering and aggregation: several state sets in a flowpipe or several successors for a jump can be over-approximated by a single set (see Section 7.1.5 below). Last but not least, the representation size is often reduced on the cost of an additional over-approximation error.

For practical applicability it is furthermore important to be able to increase the precision, i.e., to reduce the approximation error on the cost of increased computational effort. One natural method is to reduce the time-step size for the flowpipe construction. Because of its crucial impact on the tightness of the flowpipe, different approaches have been suggested in the literature to tighten the initial sets for flowpipe construction [FR09, Gir05, LG09].

7.1.5 Clustering and Aggregation

To reduce complexity, some tools apply techniques to over-approximate several sets that should be processed by a single set. This reduces the future computational effort, however, it naturally leads to less precision.

The sets that will be over-approximated stem from segments of the same flowpipe. E.g., we could consider several consecutive flowpipe segments, or the sets resulting from them after their intersection with the guard of a jump.

One method is *clustering* of sets. In this case, the considered sets are clustered into several groups and each group is over-approximated by a single set. The over-approximation can be of different representation types. For example, we can apply the convex hull operation, the result being in general a convex polyhedron. Alternatively we can apply union operation to achieve a result in the same representation. The targeted number and size of the clusters can be used to drive the balancing between efficiency and precision.

Aggregation is a special type of clustering building a single "cluster", i.e., it over-approximates all of the considered sets by a single set. Similar to clustering, the resulting over-approximation can be obtained in different representations.

Clustering and aggregation are sometimes used in *combination*: this is useful if we use two different state set representations having different computational complexities for the union computations. We can apply the computationally less expensive (but usually also less precise) representation for a clustering into several smaller groups, and use the other representation to compute a more precise over-approximation of the resulting, smaller number of sets.

In the following we have a closer look at representation by convex polyhedra.

7.2 Convex Polyhedra

Next we discuss state set representation by *convex polyhedra* in more detail. Some polyhedra are depicted in Figure 7.9

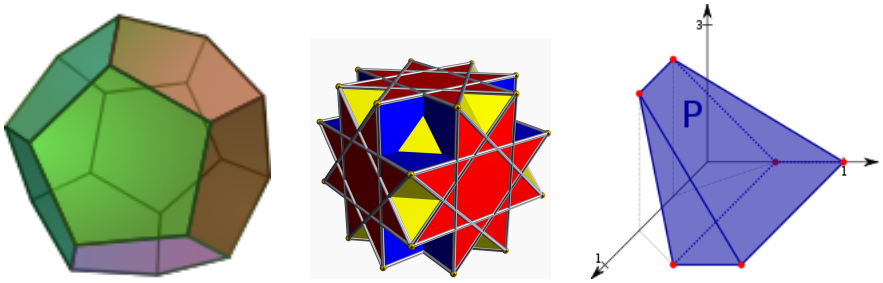


Figure 7.9: Polyhedra

Definition 7.6. *A* (convex) polyhedron *in* $\mathbb{R}^d$ *is the solution set to a finite number of inequalities with real coefficients in* d *real variables. A bounded polyhedron is called* polytope.

In the following we restrict ourselves to convex polytopes. An extension to convex polyhedra is possible, but a bit more involved.

We introduce two representation forms for convex polytopes. Defining a polytope by its facets yields an $\mathcal{H}$-representation, whereas the $\mathcal{V}$-representation stores the vertices[1].

Definition 7.7 (Closed halfspace). *A* d*-dimensional* closed halfspace *is a set* $\mathcal{H} = \{x \in \mathbb{R}^d \mid c \cdot x \leq z\}$ *for some* $c \in \mathbb{R}^d$*, called the* normal *of the halfspace, and a* $z \in \mathbb{R}$.

Definition 7.8 ($\mathcal{H}$-polyhedron, $\mathcal{H}$-polytope). *A* d*-dimensional* $\mathcal{H}$-polyhedron $P = \bigcap_{i=1}^{n} \mathcal{H}_i$ *is the intersection of finitely many closed halfspaces. A bounded* $\mathcal{H}$*-polyhedron is called an* $\mathcal{H}$-polytope.

The facets of a d-dimensional $\mathcal{H}$-polytope are $d-1$-dimensional $\mathcal{H}$-polytopes.

An $\mathcal{H}$-polytope

$$P \quad = \quad \bigcap_{i=1}^{n} \mathcal{H}_i \quad = \quad \bigcap_{i=1}^{n} \{x \in \mathbb{R}^d \mid c_i \cdot x \leq z_i\}$$

can also be written in the form

$$P = \{x \in \mathbb{R}^d \mid Cx \leq z\}.$$

We call (C, z) the $\mathcal{H}$*-representation* of the polytope. Each row c_i of C is the normal vector to the ith facet of the polytope. Note that each $\mathcal{H}$-polytope P has a finite number of vertices which we denote by $V(P)$.

Definition 7.9. *A set* S *is called* convex, *if*

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

$\mathcal{H}$-polyhedra are convex sets.

[1] $\mathcal{H}$ stays for halfspace and $\mathcal{V}$ for vertex.

Definition 7.10 (Convex hull). *Given a set $V \subseteq \mathbb{R}^d$, the* convex hull $CH(V)$ *of V is the smallest convex set that contains V.* $CH(V)$

For a finite set $V = \{v_1, \ldots, v_n\}$ the convex hull can be computed by

$$CH(V) = \{x \in \mathbb{R}^d \mid \exists \lambda_1, \ldots, \lambda_n \in [0,1] \subseteq \mathbb{R}^d.\ \sum_{i=1}^{n} \lambda_i = 1 \wedge \sum_{i=1}^{n} \lambda_i v_i = x\}.$$

Definition 7.11 ($\mathcal{V}$-polytope). *A* $\mathcal{V}$-polytope $P = CH(V)$ *is the convex hull of a finite set $V \subset \mathbb{R}^d$. We call V the* $\mathcal{V}$-representation *of the polytope.*

Note that all $\mathcal{V}$-polytopes are bounded. Note furthermore that both representations are in general not canonical as they may be non-redundant: The $\mathcal{H}$-representation may contain redundant subsumed inequations, and the $\mathcal{V}$-representation may contain redundant inner points that are not vertices. This implies that there may be different representations of a single polyhedron. Such superfluous data do not pose theoretical problems, but of course increase the effort of computations. Redundant information can be removed by solving (a set of) linear programms.

For each $\mathcal{H}$-polytope, the convex hull of its vertices defines the same set in the form of a $\mathcal{V}$-polytope, and vice versa, each set defined as a $\mathcal{V}$-polytope can be also given as an $\mathcal{H}$-polytope by computing the halfspaces defined by its facets. This is stated by Motzkin's theorem. However, the translations between the $\mathcal{H}$- and the $\mathcal{V}$-representations of polytopes can be exponential in the state space dimension d.

Given a convex polytope, the sizes of the $\mathcal{H}$- and $\mathcal{V}$-representations can strongly differ. For example, on the one hand the d-dimensional cube

$$\{x = (x_1, \ldots, x_d) \in \mathbb{R}^d \mid \forall 1 \leq i \leq d.\ -1 \leq x_i \leq 1\}$$

has $2d$ facets and 2^d vertices. On the other hand, the d-dimensional crosspolytope

$$\{x = (x_1, \ldots, x_d) \in \mathbb{R}^d \mid \sum_{i=1}^{d} |x_i| \leq 1\}$$

has $2d$ vertices and 2^d facets.

If we represent reachable sets of hybrid automata by polytopes, we again need certain *operations* on convex polytopes. In the following we discuss

- the *membership* problem,
- the *intersection*, and the
- the *union* of two polytopes.

As we will see, the computations have different complexities in the different representations. Many operations are easily solvable in one of the representation and hard in the other one and vice versa. One could think of converting polytopes for each needed operation into the representation for which the operation is cheap (indeed this is sometimes done). However, note that the conversion can have exponential costs.

- The *membership* problem can be solved in linear time in d in the $\mathcal{H}$-representation. Given an $\mathcal{H}$-polytope defined by $Cx \leq z$ and a point $p \in \mathbb{R}^d$, to check if p is contained in the polytope just substitute p for x in $Cx \leq z$ to check if the inequation holds.

For the $\mathcal{V}$-representation we have to solve a linear programming problem. Given a $\mathcal{V}$-polytope defined by the vertex set V, we have to check satisfiability of

$$\exists \lambda_1, \ldots, \lambda_n \in [0,1] \subseteq \mathbb{R}^d. \ \sum_{i=1}^{n} \lambda_i = 1 \wedge \sum_{i=1}^{n} \lambda_i v_i = x \ .$$

Alternatively we can also convert the $\mathcal{V}$-polytope into an $\mathcal{H}$-polytope by computing its facets and check membership for the $\mathcal{H}$-representation.

- The *intersection* for two polytopes P_1 and P_2 in the $\mathcal{H}$-representation is again cheap: Given an $\mathcal{H}$-polytope defined by $C_1 x \leq z_1$ and $C_2 x \leq z_2$, their intersection is represented by the $\mathcal{H}$-polytope with $\begin{pmatrix} C_1 \\ C_2 \end{pmatrix} x \leq \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$. Note that the resulting representation is in general not minimal.
  Again, the intersection computation for the $\mathcal{V}$-representation is more complex (NP-hard); we do not discuss it here. Assume two $\mathcal{V}$-polytopes P_1 and P_2 having the vertex sets V_1 respectively V_2. We can convert the polytopes to $\mathcal{H}$-polytopes, compute their intersection, and convert the result back to a $\mathcal{V}$-polytope.
- For the union, note that the union of two convex polytopes is in general not a convex polytope. The standard way to make the union computation closed under convex polytopes is to take the convex hull of the union.
  This time the computation for the $\mathcal{V}$-representation is more efficient. Assume two $\mathcal{V}$-polytopes defined by the vertex sets V_1 and V_2. The $\mathcal{V}$-representation of their union is given by $V_1 \cup V_2$. Note again that the representation is not redundant (however, it can be made minimal with additional effort).
  To compute the union of two $\mathcal{H}$-polytopes defined by $C_1 x \leq z_1$ and $C_2 x \leq z_2$ is more complex (NP-hard), and we do not handle it here. Alternatively we can convert the polytopes to $\mathcal{V}$-polytopes, compute the union, and compute back the result.

Subject index

Bibliography

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theor. Comput. Sci.*, 138(1):3–34, 1995. 8, 29, 63

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. 7

[AK12] Matthias Althoff and Bruce H. Krogh. Avoiding geometric intersection operations in reachability analysis of hybrid systems. In *Proc. of the 15th ACM Int. Conf. on Hybrid Systems: Computation and Control (HSCC'12)*, pages 45–54. ACM Press, 2012. 87

[BHZ08] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008. 83

[BK08] Ch. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008. 7, 23, 24, 39

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 3. print edition, 2001. 27

[FGD+11] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang, and Oded Maler. SpaceEx: Scalable verification of hybrid systems. In *Proc. of CAV'11*, volume 6806 of *LNCS*, pages 379–395. Springer, 2011. 84

[FHR+07] Martin Fränzle, Christian Herde, Stefan Ratschan, Tobias Schubert, and Tino Teige. Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007. 82

[FKL13] G. Frehse, R. Kateja, and C. Le Guernic. Flowpipe approximation and clustering in space-time. In *Proc. of HSCC'13*, pages 203–212. ACM, 2013. 84

[FR09] G. Frehse and R. Ray. Design principles for an extendable verification tool for hybrid systems. In *Proc. of ADHS'09*, pages 244–249. IFAC-PapersOnLine, 2009. 87

[Fre15] Goran Frehse. Reachability of hybrid systems in space-time. In *Proc. of EMSOFT'15*, pages 41–50. IEEE Press, 2015. 84

[Gir04] Antoine Girard. *Algorithmic Analysis of Hybrid Systems.* PhD thesis, Institut National Polytechnique de Grenoble - INPG, France, 2004. 80

[Gir05] Antoine Girard. Reachability of uncertain linear systems using zonotopes. In *Proc. of HSCC'05*, volume 3414 of *LNCS*, pages 291–305. Springer, 2005. 80, 87

[GJ00] Ewgenij Gawrilow and Michael Joswig. polymake: a framework for analyzing convex polytopes. In *Polytopes – combinatorics and computation (Oberwolfach, 1997)*, volume 29 of *DMV Sem.*, pages 43–73. Birkhäuser, Basel, 2000. 83

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proc. of the Eleventh Annual IEEE Symposium On Logic In Computer Science (LICS'96)*, pages 278–293, New York, USA, July 1996. IEEE Computer Society Press. 29

[HKPV98] Henzinger, Kopke, Puri, and Varaiya. What's decidable about hybrid automata? *JCSS: Journal of Computer and System Sciences*, 57, 1998. 8, 57

[Kat99] Joost-Pieter Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32-1 of *Arbeitsberichte der Informatik.* Friedrich-Alexander-Universität Erlangen Nürnberg, 1999. 27

[LG09] Colas Le Guernic. *Reachability analysis of hybrid systems with linear continuous dynamics.* PhD thesis, Université Joseph-Fourier-Grenoble I, France, 2009. 80, 81, 87

[LGG10] Colas Le Guernic and Antoine Girard. Reachability analysis of linear systems using support functions. *Nonlinear Analysis: Hybrid Systems*, 4(2):250–262, 2010. 84