

Modeling and Analysis of Hybrid Systems

Lecture Notes

Prof. Dr. Erika Ábrahám

Version: April 18, 2012

Theory of Hybrid Systems
Informatik 2

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

Contents

Notation	5
1 Introduction	7
2 Hybrid Systems and Hybrid Automata	9
2.1 Hybrid Systems	9
2.2 Labeled State Transition Systems	12
2.3 Labeled Transition Systems	17
2.4 Hybrid Automata	22
3 Discrete Systems	31
3.1 Propositional Logic	31
3.2 Temporal Logics	32
3.2.1 Linear Temporal Logic	33
3.2.2 Computation Tree Logic	35
3.2.3 CTL*	36
3.2.4 The Relation of LTL, CTL, and CTL*	37
3.3 Model Checking	38
3.4 Discrete-Time Systems	40
4 Timed Automata	43
4.1 Timed Automata	43
4.1.1 Continuous-Time Phenomena	49
4.2 Timed Temporal Logics	51
4.3 Model Checking TCTL for Timed Automata	53
4.3.1 Eliminating Timing Parameters	53
4.3.2 Finite State Space Abstraction	54
4.3.3 The Region Transition System	58
4.3.4 TCTL Model Checking	62
5 Rectangular Automata	63
5.1 Syntax and Semantics of Rectangular Automata	63
5.2 Decidability of Rectangular Automata	66
5.2.1 From Initialized Stopwatch Automata to Timed Automata	67

CONTENTS

5.2.2	From Initialized Singular Automata to Initialized Stopwatch Automata	68
5.2.3	From Initialized Rectangular Automaton to Initialized Singular Automaton	68
6	Linear Hybrid Automata	69
6.1	Syntax and Semantics	69
6.2	Forward analysis	71
6.3	Backward analysis	73
6.4	Approximate analysis	74
6.5	Minimization	75
7	General Hybrid Automata	83
7.1	Orthogonal Polyhedra	86
7.1.1	Definition of orthogonal polyhedra	86
7.1.2	Representation of orthogonal polyhedra	89
7.1.3	Membership problem	95
7.1.4	Computing the Intersection of Orthogonal Polyhedra	101
7.2	Convex Polyhedra	105
	Index	108
	References	112

Notation

We use the following standard notations:

Symbol	Description
--------	-------------

Sets and set operators

\mathbb{Z}	set of integers
\mathbb{N}	set of natural numbers including 0
$\mathbb{N}_{>0}$	set of positive natural numbers excluding 0
\mathbb{Q}	set of rational numbers
\mathbb{R}	set of real numbers
\mathbb{R}^d	the d -dimensional real space
$\mathbb{R}_{\geq 0}$	set of non-negative real numbers including 0
2^M	powerset (set of subsets) of the set M

Mappings

id	identity mapping $id : M \rightarrow M, id(m) = m$ for a set M
$f(M)$	image $\{f(m) \mid m \in M\}$ of a set M according to a mapping f

CONTENTS

Chapter 1

Introduction

This book deals with the modeling and analysis of *hybrid systems* from the view point of computer science. Hybrid systems are systems with mixed discrete-continuous behavior. Typical examples are physical systems controlled by a discrete controller. Whereas methods and tools for the modeling and simulation of the dynamic continuous behavior is hosted mainly in physics and control theory, the modeling and analysis of discrete systems is a subject of computer science. Hybrid systems combine these two parts, resulting in systems with quite complex behaviour, and posing a challenging task for their analysis.

Chapter 2 introduces *hybrid systems* on a number of examples, and describes automata-based *modeling* approaches of discrete and hybrid systems. We use labeled state transition systems to model finite-state systems, and labeled transition systems to model general discrete systems with possibly infinite state spaces. Finally, hybrid automata are introduced for the modeling of hybrid systems.

Chapter 3 first recalls the *temporal logics* LTL, CTL, and CTL* to specify properties of discrete systems, and gives a short introduction to CTL *model checking*. We give a short description of *discrete-time systems* before we deal with continuous-time systems in the next chapter.

Chapter 4 introduces *timed automata*, a simple class of hybrid automata that extends discrete systems with a notion of time. We introduce the timed temporal logic TCTL, used to specify properties of timed automata. We show that the validity of TCTL properties for timed automata is decidable by giving a model checking algorithm.

Timed automata build a quite restricted class of hybrid automata. In Chapter 5 we define rectangular automata, a bit more general class, which is at the boundary of decidability: though checking TCTL properties of rectangular automata (under some additional conditions) is a decidable problem, relaxing any of the restrictions lead to undecidability.

Though more expressive classes of hybrid automata are in general undecidable, we need them when we want to model more complex systems without strong abstraction. Though undecidability implies that we cannot give any complete model checking algorithm for them, there exist incomplete algorithms for their analysis. Such a more expressive class is

the class of *linear hybrid automata*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 for the reachability analysis of linear hybrid systems, and mention some approximation and abstraction techniques.

The reachability analysis of even more general classes of hybrid automata requires special representations of state sets, which build the content of Chapter 7. We discuss different geometrical forms like convex polyhedra, orthogonal polyhedra, oriented rectangular hulls, and zonotopes, suited for the (over-approximative) representation of state sets of general hybrid automata. Using such representations we give an incomplete fixed-point-based algorithm for their reachability analysis.

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

Chapter 2

Hybrid Systems and Hybrid Automata

In this chapter we first introduce hybrid systems in Section 2.1. Before we come to their modeling, we describe labeled state transition systems (LSTSs) suited for the modeling of discrete systems in Section 2.2. We generalize LSTS to labeled transition systems (LTSs) in Section 2.3. Finally, we define hybrid automata for the modeling of hybrid systems in Section 2.4.

2.1 Hybrid Systems

Discrete systems are systems with a countable number of states. Typical examples are circuits, digital chips, and computers (as far as we abstract away from the underlying physical processes). Though the state space of a program can be very large, it is not only countable but even finite. Thus also programs running on a computer can be seen as discrete systems if we do not care about their real-time behaviour.

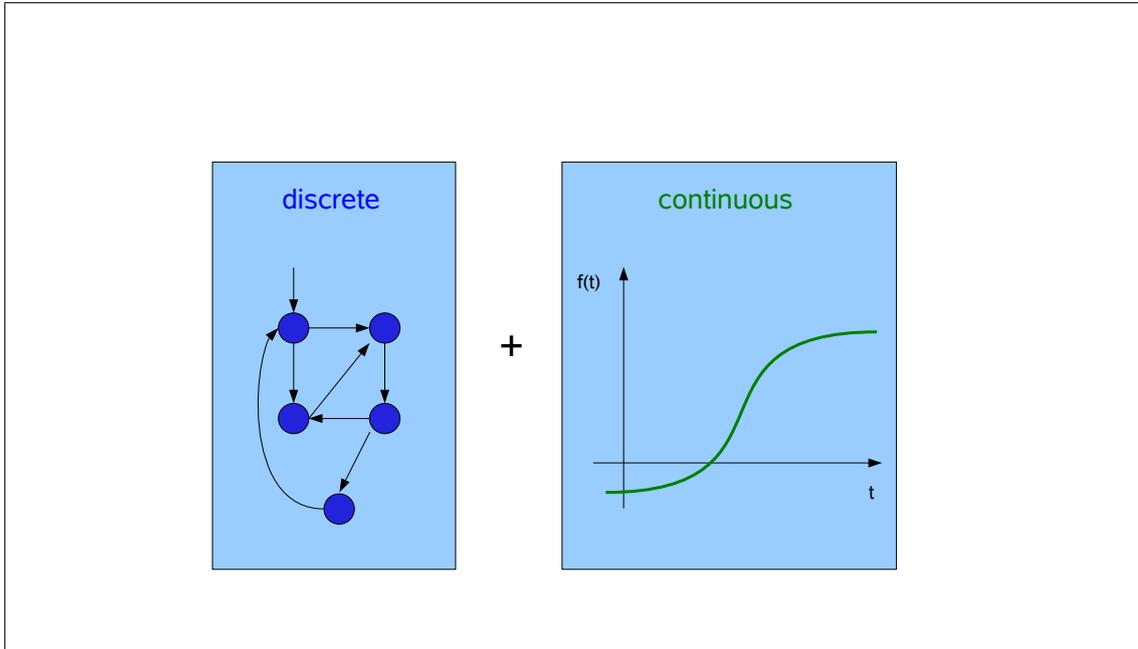
Continuous systems are systems with a continuous behaviour and a real-valued state space. Physical systems with quantities like time, temperature, speed, acceleration etc. are continuous systems. Their evolution over time can be described by continuous functions or ordinary differential equations.

Hybrid systems are systems with combined discrete and continuous behavior (cf. Figure 2.1). 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 behaviour like speed and acceleration. Behind the autopilot of an aeroplane there is a program running on a computer and acting with the physical environment. Some examples of discrete, continuous, and hybrid systems are illustrated on Figure 2.2.

We introduce some hybrid system examples from [ACH⁺95, Hen96].

Example 1 (Thermostat).

Assume a thermostat, which senses the temperature x of a room and turns a heater on


Figure 2.1: Hybrid systems

and off in order to keep the temperature between 18°C and 22°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 is a constant of the heater and K a room constant. If the temperature is 21°C or above, but at latest when it reaches 22°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 19°C or below, but at latest if it reaches 18°C , the heater gets switched on. Figure 2.3 visualizes a possible behaviour of the system.

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 behaviour.

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.

Example 2 (Water-level monitor).

Assume two constantly leaking water tanks and a hose that refills exactly one of the tanks at each point in time (cf. Figure 2.4). 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 the levels $\dot{x}_1 = -v_1$ and $\dot{x}_2 = -v_2$ for some $v_1, v_2 \in \mathbb{R}^+$ without refilling. The hose fills $w \in \mathbb{R}^+$ 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. Refilling the second tank yields $\dot{x}_2 = w - v_2$.



Figure 2.2: Discrete, continuous, and hybrid systems

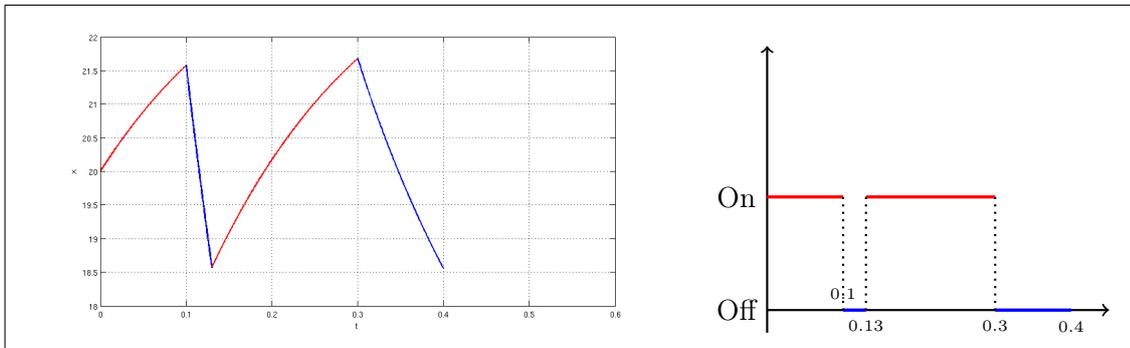


Figure 2.3: A possible behavior of the thermostat

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 (Bouncing ball).

Assume a bouncing ball with the initial height $x_1 \geq 0$ and with an initial upwards directed speed $x_2 > 0$. Due to gravity, the ball has the acceleration $\dot{x}_2 = -g$. 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 position $x_1 = 0$ with a speed $x_2 < 0$. When bouncing, the sign of x_2 gets inverted, and a part of the ball's kinetic energy gets lost. Its speed after bouncing is $-cx_2$ with some $c \in (0, 1) \subseteq \mathbb{R}$ and x_2 the speed before bouncing. Figure 2.5 illustrates the behaviour 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 behaviour, 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.

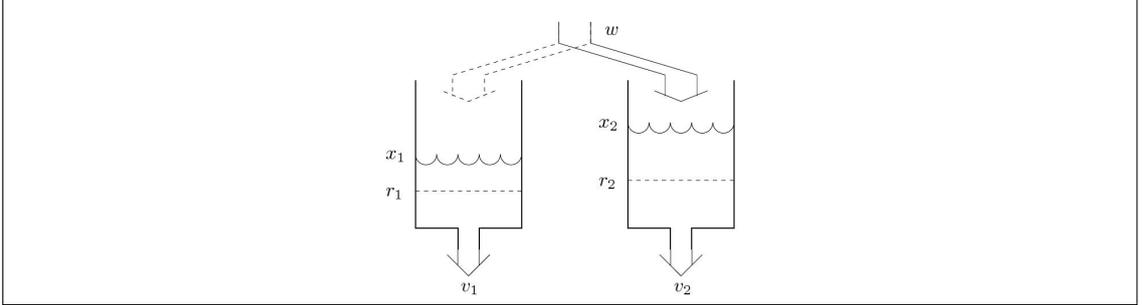


Figure 2.4: Water-level monitor

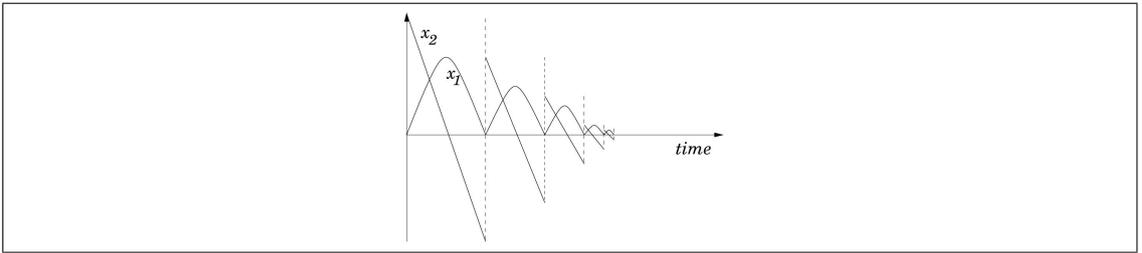


Figure 2.5: A possible behavior of the bouncing ball

Hybrid systems are often modeled as *hybrid automata*. Before giving a definition of hybrid automata, we recall the definitions of state transition systems and transition systems.

2.2 Labeled State Transition Systems

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 1 (Syntax of labeled state transition systems).

- A labeled state transition system (LSTS) is a tuple $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$ with
- a (possibly infinite) state set Σ ,
 - a (synchronization) label set Lab ,
 - a set of transitions $Edge \subseteq \Sigma \times Lab \times \Sigma$, and
 - a non-empty set of initial states $Init \subseteq \Sigma$.

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

Definition 2 (Semantics of LSTS).

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

$$\frac{(\sigma, a, \sigma') \in Edge}{\sigma \xrightarrow{a} \sigma'} \text{Rule}_{discrete}$$

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 $\pi = \sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \dots$ with

π

- $\sigma_0 \in \text{Init}$ and
- $\sigma_i \xrightarrow{a_i} \sigma_{i+1}$ for all $i \geq 0$ for infinite paths and for all $0 \leq i < |\pi|$ with $|\pi|$ the number of steps in the path for finite paths.

A state $\sigma \in \Sigma$ is called reachable iff there is a finite path leading to it.

The labels of the set Lab are attached to edges and are used for synchronization purposes in the parallel composition.

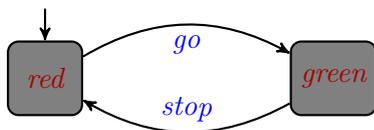
To be able to formalize properties of LSTSs, it is common to define a set of atomic propositions AP and a labeling function $L : \Sigma \rightarrow 2^{AP}$ assigning a set of atomic propositions to each state. The set $L(\sigma)$ consists of all propositions that are defined to hold in σ . These propositional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

AP, L

A labeled state transition system $\mathcal{LSTS} = (\Sigma, \text{Lab}, \text{Edge}, \text{Init})$ 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 4 (Pedestrian light).

We model a pedestrian traffic light in a crossing by a labeled state transition system $\mathcal{LSTS} = (\Sigma, \text{Lab}, \text{Edge}, \text{Init})$. 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 $\Sigma = \{\text{red}, \text{green}\}$. Assume the light is initially red, i.e., $\text{Init} = \{\text{red}\}$. Possible state changes go from red to green and from green to red, yielding $\text{Edge} = \{(\text{red}, \text{go}, \text{green}), (\text{green}, \text{stop}, \text{red})\}$ for a possible label set $\text{Lab} = \{\text{go}, \text{stop}\}$. 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 not only deterministic, but has even only a single run $\text{red} \xrightarrow{\text{go}} \text{green} \xrightarrow{\text{stop}} \text{red} \dots$

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

Definition 3 (Parallel composition of LSTS).

Let

$$\mathcal{LSTS}_1 = (\Sigma_1, Lab_1, Edge_1, Init_1) \text{ and}$$

$$\mathcal{LSTS}_2 = (\Sigma_2, Lab_2, Edge_2, Init_2)$$

$\mathcal{LSTS}_1 ||$
 \mathcal{LSTS}_2

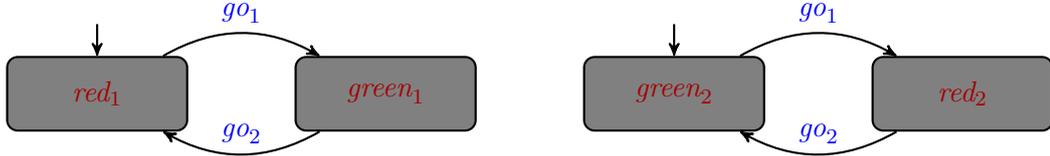
be two LSTSs. The parallel composition $\mathcal{LSTS}_1 || \mathcal{LSTS}_2$ is the LSTS $(\Sigma, Lab, Edge, Init)$ with

- $\Sigma = \Sigma_1 \times \Sigma_2$,
- $Lab = Lab_1 \cup Lab_2$,
- $((s_1, s_2), a, (s'_1, s'_2)) \in Edge$ 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. $a \in Lab_1 \setminus Lab_2$, $(s_1, a, s'_1) \in Edge_1$, and $s_2 = s'_2$, or
 3. $a \in Lab_2 \setminus Lab_1$, $(s_2, a, s'_2) \in Edge_2$, 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 5 (Two pedestrian lights).

Assume now a crossing of two roads with two pedestrian lights, similar to those from Example 4, 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 $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init) = \mathcal{LSTS}_1 || \mathcal{LSTS}_2$ is by definition:

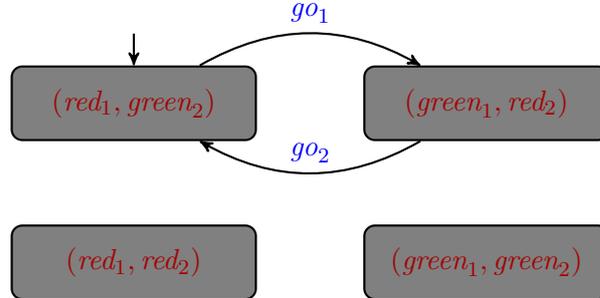
$$\Sigma = \{(green_1, green_2), (green_1, red_2), (red_1, red_2), (red_1, green_2)\}$$

$$Lab = \{go_1, go_2\}$$

$$Edge = \{((red_1, green_2), go_1, (green_1, red_2)), ((green_1, red_2), go_2, (red_1, green_2))\}$$

$$Init = \{(red_1, green_2)\}$$

The parallel composition can be visualized by 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 even a single run $(red_1, green_2) \xrightarrow{go_1} (green_1, red_2) \xrightarrow{go_2} (red_1, green_2) \dots$

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

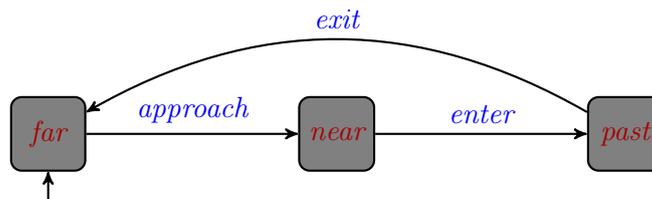
Example 6 (Railroad crossing).

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.

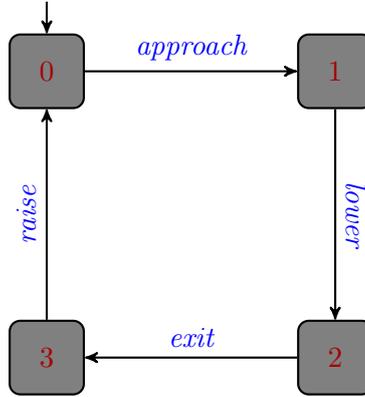
- Sensors recognize if the train is approaching to the gate and an “approach” signal is sent to the controller. Similarly, if the train has left the railroad crossing, an “exit” signal gets sent to the controller.
- The controller reacts to an incoming “approach” signal with the sending of a “lower” signal to the gate. The “exit” signal from the train triggers 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:

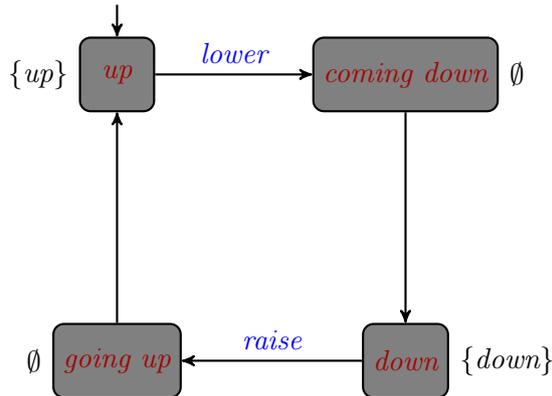
- Train



- *Controller*



- *Gate*



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 $(\mathcal{H}_{Train} \parallel \mathcal{H}_{Controller}) \parallel \mathcal{H}_{Gate}$ is the content of Exercise 4. 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 into the state 1 to be able to synchronize on it. Thus first synchronization on $approach$ must take place. Thus each path of the composition starts with the step $(far, 0, up) \xrightarrow{approach} (near, 1, up)$.

2.3 Labeled Transition Systems

If an LTS has a large or even infinite state set, it is usually more convenient to define it as a transition system having a finite set Loc of *locations* which can possibly be reached by different states of the system. In order to keep track of the current states in the locations, *variables* are introduced. Here we are interested in 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 $\nu : Var \rightarrow \mathbb{R}$ assigning values to the variables. We use V_{Var} (or short V) to denote the set of all valuations for the variable set Var . The system's state is determined by the current location and the current valuation. Therefore *states* $\sigma = (l, \nu)$ are location-valuation pairs from the set $\Sigma = Loc \times V$.

A *labeled transition system (LTS)* has a set of locations. The locations are connected by discrete labeled transitions (edges) with guards and effects. The guard and effect of a transition is specified in form of a transition relation $\mu \subseteq V \times V$: the transition can be taken with an evaluation ν thereby changing the evaluation to ν' iff $(\nu, \nu') \in \mu$. A set of initial states specify the states in which the execution may start.

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” variables of an LTS whose values may not be changed by non-synchronizing steps of other parallel LTSs.

Definition 4 (Syntax of labeled transition systems).

A labeled transition system (LTS) is a tuple $\mathcal{LTS} = (Loc, Var, Con, Lab, Edge, Init)$ with

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

Definition 5 (Semantics of LTS).

The semantics of $\mathcal{LTS} = (Loc, Var, Con, Lab, Edge, Init)$ is given by an operational semantics with the following single rule:

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

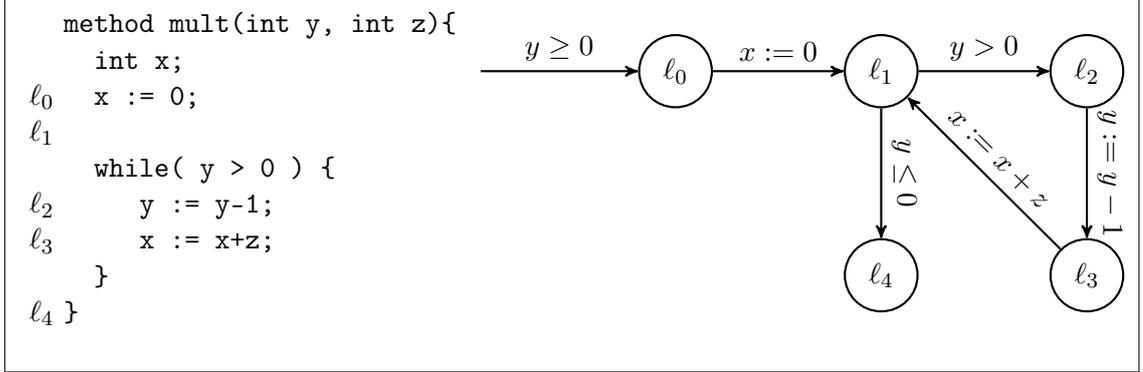


Figure 2.6: Modeling a simple while program with an LTS.

π

A path (or run or execution) π of \mathcal{LTS} is a sequence $\sigma_0 \xrightarrow{a_0} \sigma_1 \xrightarrow{a_1} \sigma_2 \dots$ of states $\sigma_i \in \Sigma$ with $\sigma_0 \in \text{Init}$ and $\sigma_i \xrightarrow{a_i} \sigma_{i+1}$ for all i . A state is called reachable in an \mathcal{LTS} iff there is a path of the \mathcal{LTS} leading to it.

Based on the operational semantics, an LTS induces an underlying LSTS state space model: a transition $(l, \nu) \xrightarrow{a} (l', \nu')$ 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 next example shows how an LTS can be used to describe program flows.

Example 7 (Modeling a simple while program).

The simple while program of Figure 2.6 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 $\mathcal{LTS} = (\text{Loc}, \text{Var}, \text{Con}, \text{Lab}, \text{Edge}, \text{Init})$ where

- $\text{Loc} = \{\ell_0, \ell_1, \ell_2, \ell_3, \ell_4\}$,
- $\text{Var} = \{x, y, z\}$,
- $\text{Con}(l) = \text{Var}$ for each $l \in \text{Loc}$,
- $\text{Lab} = \{\tau\}$,
- $V = \{\nu : \text{Var} \rightarrow \mathbb{Z}\}$,
- $\text{Edge} =$

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

- $Init = \{(\ell_0, \nu) \mid \nu(y) \geq 0\}$

with $\tau_l = \{(\nu, \nu') \in V^2 \mid \nu = \nu'\}$ for all $l \in Loc$.

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. If a step is synchronizing or not depends on the fact if both systems have the step's label in their label sets. That means, for a label being in the label set of both components, one of the components can take a transition with this label only if the other component also takes a transition with this 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 execute a 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. Those variables that a components has under its control and may not be modified during its τ -transitions are defined by the function Con .

Definition 6 (Parallel composition of LTSs).

Let

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

be two LTSs. The parallel composition or product $\mathcal{LTS}_1 \parallel \mathcal{LTS}_2$ of \mathcal{LTS}_1 and \mathcal{LTS}_2 is defined to be the LTS

$$\mathcal{LTS} = (Loc, Var, Con, Lab, Edge, Init)$$

with

- $Loc = Loc_1 \times Loc_2$,

- $Con((l_1, l_2)) = Con_1(l_1) \cup Con_2(l_2)$,
- $Lab = Lab_1 \cup Lab_2$,
- $((l_1, l_2), a, \mu, (l'_1, l'_2)) \in Edge$ iff
 - there exist $(l_1, a_1, \mu_1, l'_1) \in Edge_1$ and $(l_2, a_2, \mu_2, l'_2) \in Edge_2$ such that
 - either $a_1 = a_2 = a$ or
 $a_1 = a \in Lab_1 \setminus Lab_2$ and $a_2 = \tau$, or
 $a_1 = \tau$ and $a_2 = a \in Lab_2 \setminus Lab_1$, and
 - $\mu = \mu_1 \cap \mu_2$, and
- $Init = \{(l_1, l_2), \nu \mid (l_1, \nu) \in Init_1 \wedge (l_2, \nu) \in Init_2\}$.

Example 8.

Assume the parallel composition of the following two LTSs:

$$\mathcal{LTS}_1 = (Loc_1, Var, Con_1, Lab_1, Edge_1, Init_1)$$

$$\mathcal{LTS}_2 = (Loc_2, Var, Con_2, Lab_2, Edge_2, Init_2)$$

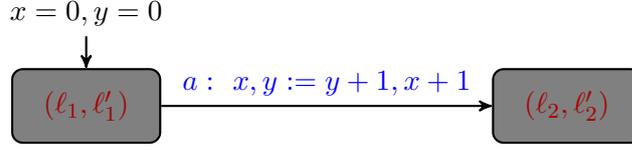
with

- $Loc_1 = \{l_1, l_2\}$, $Loc_2 = \{l'_1, l'_2\}$,
- $Var = \{x, y\}$,
- $Con_1(l_1) = Con_1(l_2) = \{x\}$, $Con_2(l'_1) = Con_2(l'_2) = \{y\}$,
- $Lab_1 = Lab_2 = \{a, \tau\}$,
- $Edge_1 = \{(l_1, a, \{(\nu, \nu') \in V^2 \mid \nu'(x) = \nu(y) + 1\}, l_2), \tau_{l_1}, \tau_{l_2}\}$,
 $Edge_2 = \{(l'_1, a, \{(\nu, \nu') \in V^2 \mid \nu'(y) = \nu(x) + 1\}, l'_2), \tau'_{l'_1}, \tau'_{l'_2}\}$,
- $Init_1 = \{(l_1, \{\nu \in V \mid \nu(x) = 0\})\}$, $Init_2 = \{(l'_1, \{\nu \in V \mid \nu(y) = 0\})\}$

with $\tau_l = \{(\nu, \nu') \in V^2 \mid \forall v \in Con_i(l). \nu(v) = \nu'(v)\}$ for all $i = 1, 2$ and $l \in Loc_i$.
 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:

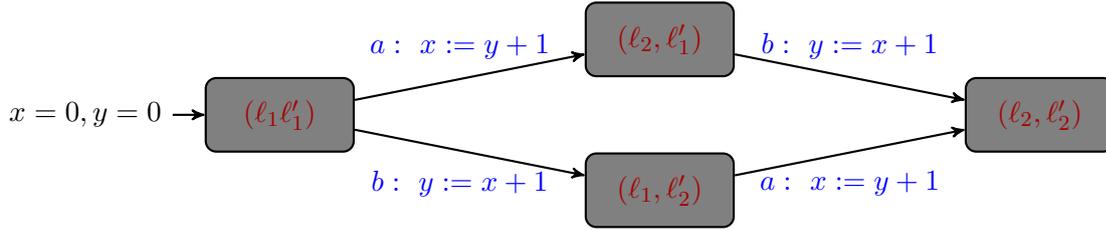


As the only non- τ -transitions of the LTSs are synchronized by the label a , all runs of the system are of the form $\sigma_0 \xrightarrow{\tau} \dots \sigma_0 \xrightarrow{a} \sigma_1 \xrightarrow{\tau} \sigma_1 \dots$ with $\sigma_0(x) = \sigma_0(y) = 0$ and $\sigma_1(x) = \sigma_1(y) = 1$.

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
 - $\sigma_0 = ((l_1, l'_1), \nu_0)$, $\nu_0(x) = \nu_0(y) = 0$,
 - $\sigma_1 = ((l_2, l'_1), \nu_1)$, $\nu_1(x) = 1$, $\nu_1(y) = 0$, and
 - $\sigma_2 = ((l_2, l'_2), \nu_2)$, $\nu_2(x) = 1$, $\nu_2(y) = 2$, or
- $\sigma_0 \xrightarrow{b} \sigma_1 \xrightarrow{a} \sigma_2$ with
 - $\sigma_0 = ((l_1, l'_1), \nu_0)$, $\nu_0(x) = \nu_0(y) = 0$,
 - $\sigma_1 = ((l_1, l'_2), \nu_1)$, $\nu_1(x) = 0$, $\nu_1(y) = 1$, and
 - $\sigma_2 = ((l_2, l'_2), \nu_2)$, $\nu_2(x) = 2$, $\nu_2(y) = 1$.

In the next section we extend labeled transition systems to hybrid automata.

2.4 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”.

Definition 7 (Syntax of hybrid automata).

- \mathcal{H} 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 $\nu : Var \rightarrow \mathbb{R}$, and $\Sigma = Loc \times V$ the set of states,
 - Act • Act is a function assigning a set of activities $f : \mathbb{R}_{\geq 0} \rightarrow V$ 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 $\sigma_0 = (l_0, \nu_0) \in Init$ from the initial set. The invariant $Inv(l_0)$ of the initial location l_0 must be satisfied by the initial valuation ν_0 , i.e., $\nu_0 \in Inv(l_0)$ 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 $f : \mathbb{R}_{\geq 0} \rightarrow V$ from $Act(l_0)$. 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 $Inv(l_0)$ of l_0 is satisfied. I.e., t time can pass by only if $\forall 0 \leq t' \leq t$ we have $f(t') \in Inv(l_0)$.

2. A discrete state change can happen if there is an enabled edge from l_0 , i.e., if there is a $(l_0, a, \mu, l_1) \in Edge$ 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., $\nu_1 \in Inv(l_1)$ 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 8 (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

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

2. Time step semantics

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

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 execution) π of \mathcal{H} is a sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \dots$ with $\sigma_0 = (l_0, \nu_0) \in Init$, $\nu_0 \in Inv(l_0)$, and $\sigma_i \rightarrow \sigma_{i+1}$ for all $i \geq 0$. A state is said to be reachable in \mathcal{H} iff there is a run of \mathcal{H} leading to it.

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 reachable states is in general uncountable, as time progress leads to continuous behaviour.

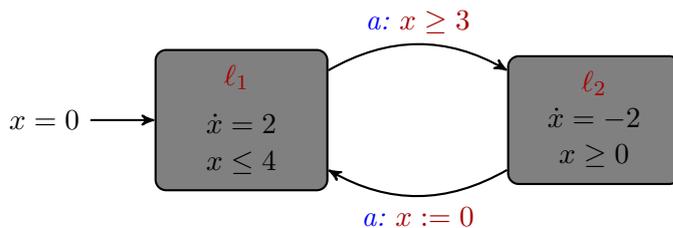
Usually, the activities of a hybrid automaton are given implicitly by differential equations, the activities being their solutions. E.g., $\dot{x} = 1$ specifies a set of activities $f : \mathbb{R}_{\geq 0} \rightarrow V$ with $f(t)(x) = t + c$ for any $c \in \mathbb{R}$ being the value of x at time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

Furthermore, valuation sets like the invariants of the locations are usually specified by formulae of the first-order logic over the reals (without quantifiers). E.g., $x > 0$ specifies the valuation set $\{\nu \in V \mid \nu(x) > 0\}$.

Finally, similarly to LTSs, also hybrid automata are often given in a graphical representation. We illustrate 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 9.

Assume the following graphical visualization of a hybrid automaton:



The formal definition is as follows:

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

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

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

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:

$$\frac{\nu(x) \geq 3 \quad \nu'(x) = \nu(x) \quad (\nu'(x) \geq 0)}{(l_1, \nu) \xrightarrow{a} (l_2, \nu')} \text{Rule}_{discrete}^{l_1 \rightarrow l_2}$$

$$\frac{\nu'(x) = 0 \quad (\nu'(x) \leq 4)}{(l_2, \nu) \xrightarrow{a} (l_1, \nu')} \text{Rule}_{discrete}^{l_2 \rightarrow l_1}$$

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:

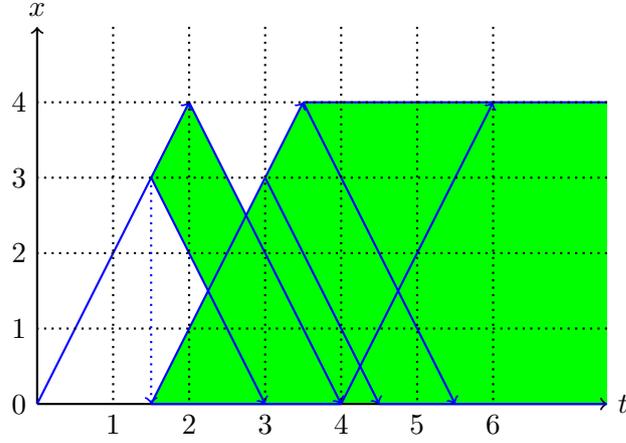
$$\frac{l \in Loc}{(l, \nu) \xrightarrow{\tau} (l, \nu)} \text{Rule}_{discrete}^{\tau}$$

For the time steps we have the following rule instances:

$$\frac{\nu'(x) \leq 4 \quad t \geq 0 \quad \nu'(x) = \nu(x) + 2t}{(l_1, \nu) \xrightarrow{a} (l_1, \nu')} \text{Rule}_{time}^{l_1}$$

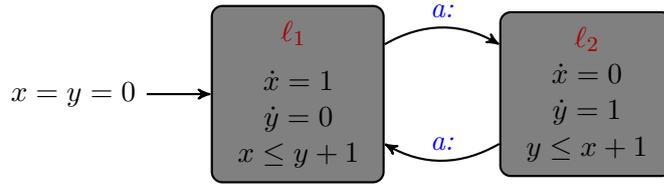
$$\frac{\nu'(x) \geq 0 \quad t \geq 0 \quad \nu'(x) = \nu(x) - 2t}{(l_2, \nu) \xrightarrow{a} (l_2, \nu')} \text{Rule}_{time}^{l_2}$$

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



Example 10.

Assume another hybrid automaton:



The formal definition is as follows:

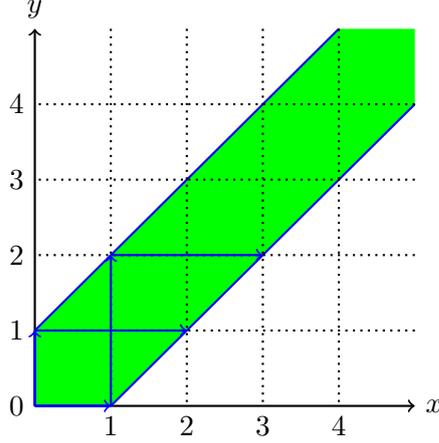
- $Loc = \{\ell_1, \ell_2\}$,
- $Var = \{x, y\}$,
- $Con(\ell_1) = Con(\ell_2) = \{x, y\}$,
- $Lab = \{\tau, a\}$,
- $Edge =$

$$\{ (\ell_1, a, \{(\nu, \nu') \in V^2 \mid \nu = \nu'\}, \ell_2), \\ (\ell_2, a, \{(\nu, \nu') \in V^2 \mid \nu = \nu'\}, \ell_1), \\ (\ell_1, \tau, \{(\nu, \nu') \in V^2 \mid \nu = \nu'\}, \ell_1), \\ (\ell_2, \tau, \{(\nu, \nu') \in V^2 \mid \nu = \nu'\}, \ell_2) \},$$

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

- $Inv(\ell_1) = \{\nu \in V \mid \nu(x) \leq \nu(y) + 1\}$,
 $Inv(\ell_2) = \{\nu \in V \mid \nu(y) \leq \nu(x) + 1\}$,
- $Init = \{(\ell_1, \nu) \in \Sigma \mid \nu(x) = 0 \wedge \nu(y) = 0\}$.

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



Example 11 (Thermostat).

Assume again the thermostat from Example 1. The modeling hybrid automaton is depicted on Figure 2.7.

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 $x \leq 22$ assures that the heater turns off at latest when the temperature reaches 22°C . Analogously for the location ℓ_{off} , where the heater is off.

Control may move from location ℓ_{on} to ℓ_{off} , switching the heater off, if the temperature is at least 21°C , and from ℓ_{off} to ℓ_{on} if the temperature is at most 19°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 21°C and 22°C , both time progress and switching the heater off are possible.

Example 12 (Water-level monitor).

The hybrid automaton model for the water-level monitor Example 2 is depicted in Figure 2.8.

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 loses 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

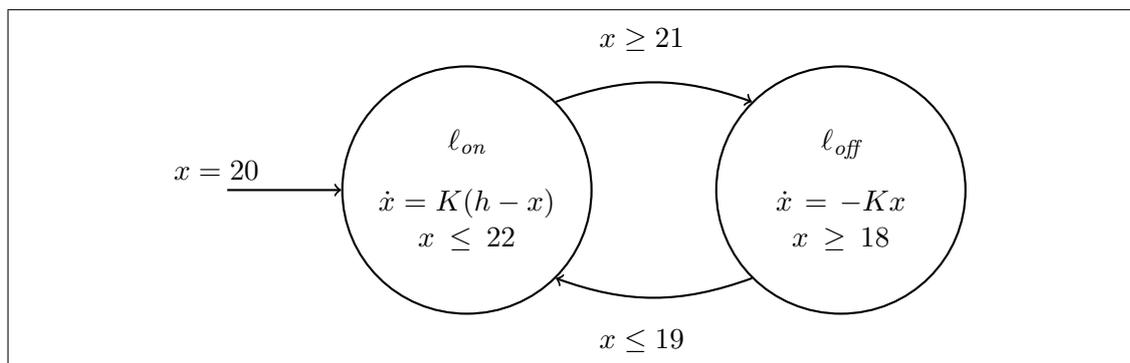


Figure 2.7: The hybrid automaton model of the thermostat

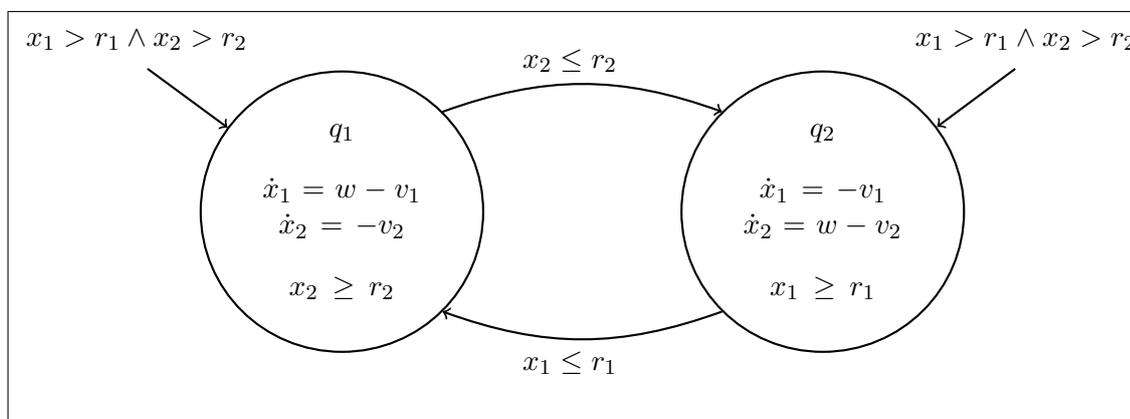


Figure 2.8: The hybrid automaton model of the water-level monitor

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 13 (Bouncing Ball).

The hybrid automaton model of the bouncing ball from Example 3 is depicted on Figure 2.9. 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.

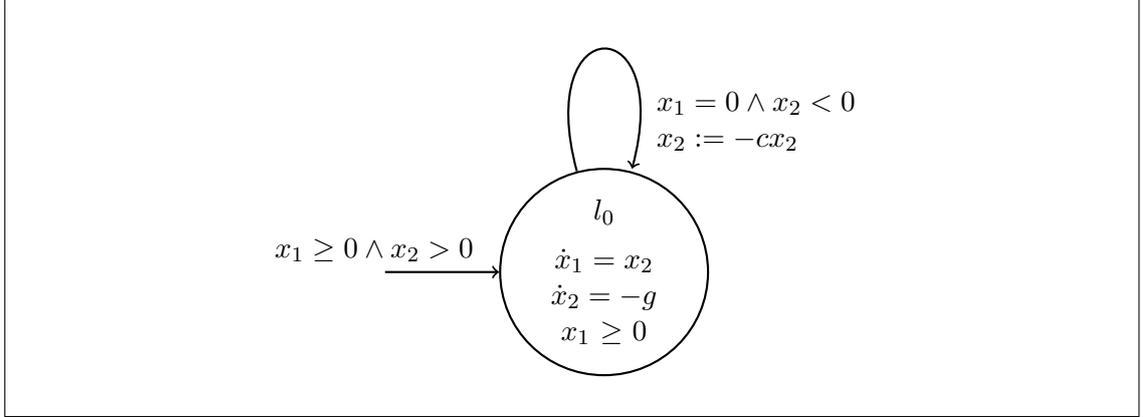


Figure 2.9: The hybrid automaton model of the bouncing ball

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 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 9 (Parallel composition of hybrid automata).

Let

$$\mathcal{H}_1 = (\text{Loc}_1, \text{Var}, \text{Con}_1, \text{Lab}_1, \text{Edge}_1, \text{Act}_1, \text{Inv}_1, \text{Init}_1) \text{ and}$$

$$\mathcal{H}_2 = (\text{Loc}_2, \text{Var}, \text{Con}_2, \text{Lab}_2, \text{Edge}_2, \text{Act}_2, \text{Inv}_2, \text{Init}_2)$$

$\mathcal{H}_1 || \mathcal{H}_2$

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 to be the hybrid automaton

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

with

- The LTS part $(\text{Loc}, \text{Var}, \text{Con}, \text{Lab}, \text{Edge}, \text{Init})$ equals the parallel composition

$$(\text{Loc}_1, \text{Var}, \text{Con}_1, \text{Lab}_1, \text{Edge}_1, \text{Init}_1) || (\text{Loc}_2, \text{Var}, \text{Con}_2, \text{Lab}_2, \text{Edge}_2, \text{Init}_2)$$

of the LTS parts of the components,

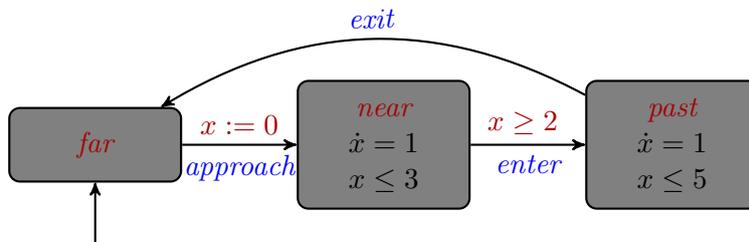
- $Act(l_1, l_2) = Act_1(l_1) \cap Act_2(l_2)$ for all $(l_1, l_2) \in Loc$, and
- $Inv(l_1, l_2) = Inv_1(l_1) \cap Inv_2(l_2)$ for all $(l_1, l_2) \in Loc$.

As a simple demonstration of the parallel composition we consider the modeling of a railroad crossing.

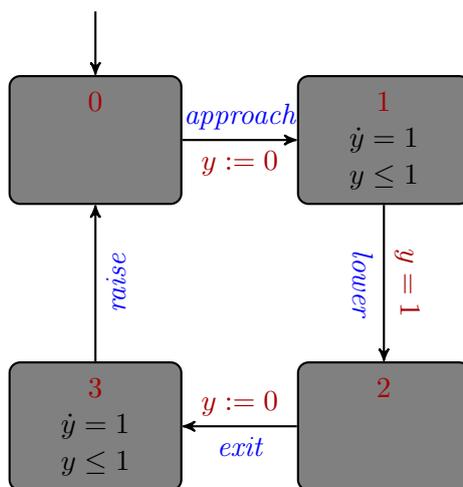
Example 14 (Railroad crossing).

We extend Example 6 with real-time behaviour as follows:

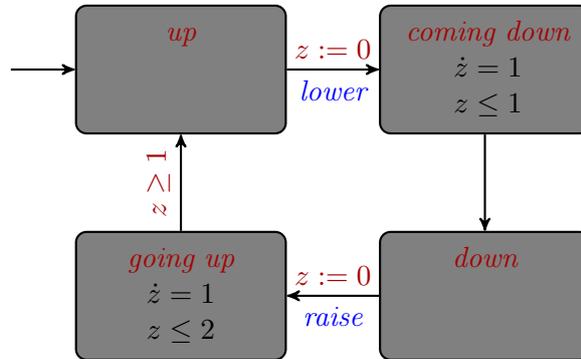
- After the train triggers the “approach” signal it reaches the gate between 2 and 3 minutes. It passes the track between the “approach” and the “exit” sensors within 5 minutes. The resulting automaton \mathcal{H}_{Train} looks as follows, with the single control variable x in each location:



- 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 resulting automaton $\mathcal{H}_{Controller}$, with y being the only control variable in each location, is as follows:



- The gate needs at most one minute to be lowered and between one and two minutes to be raised. The so obtained automaton, having z as single control variable in each location, is denoted by \mathcal{H}_{Gate} :



Exercises

Exercise 1.

Show that the LSTS parallel composition is commutative and associative.

Exercise 2.

Show that the LTS parallel composition is commutative and associative.

Exercise 3.

Show that the parallel composition of hybrid automata is commutative and associative.

Exercise 4.

Construct the automaton $(\mathcal{H}_{Train} \parallel \mathcal{H}_{Controller}) \parallel \mathcal{H}_{Gate}$ from Example 14.

Chapter 3

Discrete Systems

In the next chapters we address the analysis of hybrid systems modeled by hybrid automata. Before doing so, in this chapter we first give some preliminaries in *logics* that allow to describe properties of *discrete* systems (here on the example of labeled state transition systems). After introducing propositional logic in a nutshell in Section 3.1 we deal with temporal logics in Section 3.2. We explain the basic idea of *model checking* of discrete systems in Section 3.3. We close the chapter with discussing a discrete model of time in Section 3.4.

3.1 Propositional Logic

Assume a set of states Σ , a set of atomic propositions AP , and a labeling function $L : \Sigma \rightarrow 2^{AP}$ 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 is built up from propositions and Boolean operators, and has the abstract syntax

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

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 the Boolean operators \vee (“or”), \rightarrow (“implies”), \leftrightarrow (“if and only if”), ... can be introduced. We often omit parentheses with the convention that the strength of binding is in the order $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$, i.e., \neg binds the strongest and \leftrightarrow the weakest. We use $Form_{prop}^{AP}$ (or short $Form_{prop}$) to denote the set of all propositional formulae over the atomic proposition set AP .

Propositional 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 formulae as follows:

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

\wedge, \neg

$\vee, \rightarrow, \leftrightarrow$

$Form_{prop}$

\models_{prop}

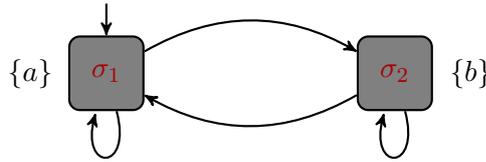
Though propositional logic is well-suited to describe *states* of a system, we are also interested in describing *computations* of systems. Propositional temporal logics, i.e., propositional logic extended with temporal modalities, can be used for this purpose.

3.2 Temporal Logics

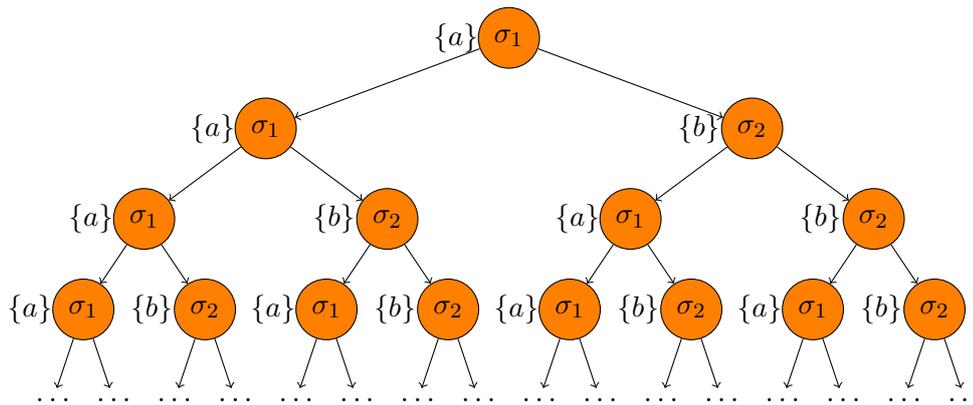
Assume in the following a labeled state transition system $\mathcal{LSTS} = (\Sigma, Lab, Edge, Init)$, a set of atomic propositions AP , and a labeling function $L : \Sigma \rightarrow 2^{AP}$. The labeled state transition system's semantics specifies the paths of the system which can be seen as a forest: for each initial state there is a tree with that initial state as root, and each state being a node in a tree has all possible successor states as children. Each path of the system corresponds to a path in one of the trees. In the following we assume deadlock-free systems and infinite paths.

Example 15 (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.

3.2.1 Linear Temporal Logic

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

Definition 10 (Syntax of LTL).

Assume a set AP of atomic propositions. LTL has the abstract syntax

\mathcal{X}, \mathcal{U}

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

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

$Form_{LTL}$

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

For a path $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots$ let $\pi(i)$ denote the i th state in π , i.e., σ_i , and let π^i denote the suffix $\sigma_i \rightarrow \sigma_{i+1} \rightarrow \dots$

Using the “next time” 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 $\pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots$ 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}

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

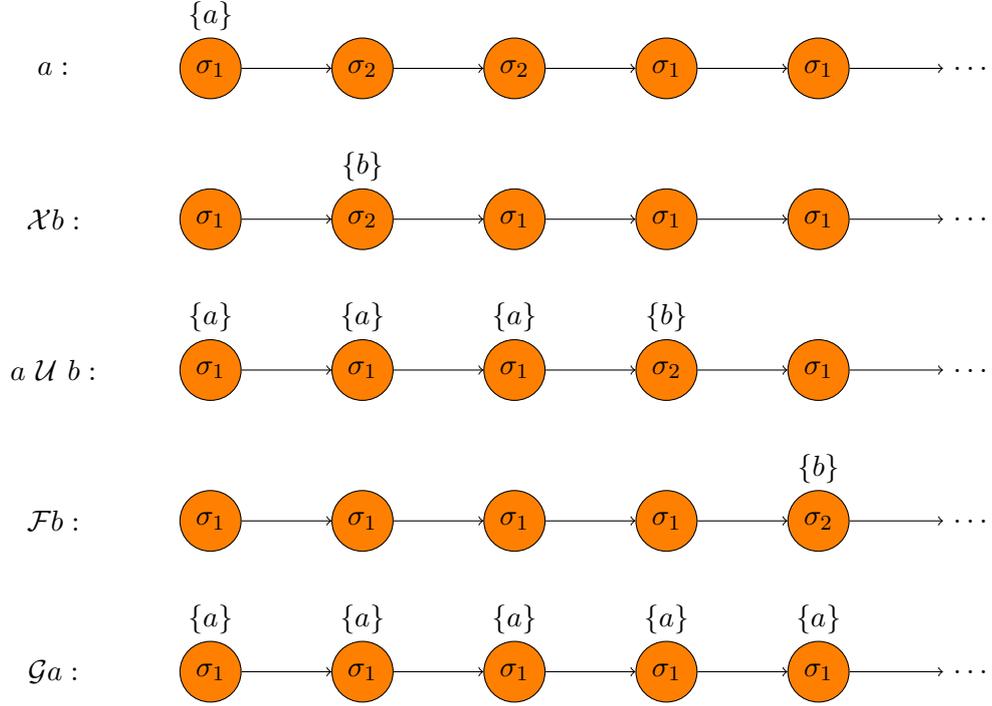
\circ, \diamond, \square

\mathcal{U}	for	\mathcal{U}
\circ	for	\mathcal{X}
\diamond	for	\mathcal{F}
\square	for	\mathcal{G}

For example, the formula $\mathcal{G}\mathcal{F}\varphi$ can also be written as $\square\diamond\varphi$.

Example 16.

We give some example LTL formulae and some paths of the system from Example 15 satisfying them. Thereby we omit labelings irrelevant for the satisfaction.


Definition 11 (Semantics of LTL).
 \models_{LTL}

The semantics of LTL is given by the satisfaction relation \models_{LTL} (or short \models) which evaluates LTL formulae in the context of a path as follows:

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

For a state transition system \mathcal{LSTS} and an LTL formula φ we define $\mathcal{LSTS} \models_{LTL} \varphi$ to hold if and only if $\pi \models_{LTL} \varphi$ for all paths π of \mathcal{LSTS} .

Example 17.

The \mathcal{LSTS} from Example 15 does not satisfy $\mathcal{F}b$, since there is a path $\pi = \sigma_1 \rightarrow \sigma_1 \rightarrow \dots$ on which b never holds. But it satisfies $\mathcal{F}a$, since the proposition a holds in the initial state.

A typical property expressible in LTL is the *liveness* property of reactive systems, expressed by formulae of the form $\mathcal{GF}\varphi$. Another class of properties of the form $\mathcal{G}\varphi$ expresses *safety*, stating that φ holds all the time.

3.2.2 Computation Tree Logic

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 12 (Syntax of CTL).

CTL state formulae over a set AP of atomic propositions can be built according to the abstract grammar

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

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

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

where ψ are CTL state formulae. 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” has to be modified since it does not follow the CTL syntax (see Exercise 5).

Also the quantifiers A and E have an alternative notation, as introduced for LTL for the temporal operators. In this alternative notation we write

\exists, \forall

\mathcal{U}	for	\mathcal{U}
\bigcirc	for	\mathcal{X}
\diamond	for	\mathcal{F}
\square	for	\mathcal{G}
\exists	for	E
\forall	for	A

For example, the formula $AGE\mathcal{F}\varphi$ can also be written as $\forall \square \exists \diamond \varphi$.

\models_{CTL}

Definition 13 (Semantics of CTL).

The satisfaction relation $\models_{CTL} \subseteq \Sigma \times Form_{CTL}$ (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:

$$\begin{aligned}
 \sigma \models_{CTL} a & \quad \text{iff } a \in L(\sigma) \\
 \sigma \models_{CTL} \psi_1 \wedge \psi_2 & \quad \text{iff } \sigma \models_{CTL} \psi_1 \text{ and } \sigma \models_{CTL} \psi_2 \\
 \sigma \models_{CTL} \neg\psi & \quad \text{iff } \sigma \not\models_{CTL} \psi \\
 \sigma \models_{CTL} E\varphi & \quad \text{iff } \pi \models_{CTL} \varphi \text{ for some } \pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots \text{ with } \sigma_0 = \sigma \\
 \sigma \models_{CTL} A\varphi & \quad \text{iff } \pi \models_{CTL} \varphi \text{ for all } \pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots \text{ with } \sigma_0 = \sigma \\
 \\
 \pi \models_{CTL} \mathcal{X}\psi & \quad \text{iff } \pi(1) \models_{CTL} \psi \\
 \pi \models_{CTL} \psi_1 \mathcal{U} \psi_2 & \quad \text{iff } \text{exists } 0 \leq j \text{ with } \pi(j) \models_{CTL} \psi_2 \text{ and } \pi(i) \models_{CTL} \psi_1 \text{ for all } 0 \leq i < j.
 \end{aligned}$$

Example 18.

For our \mathcal{LSTS} from Example 15 the CTL formula $AGEXb$ holds, since at each node of the computation tree we can take a transition into σ_2 labeled with b .

The formula $AFEGa$ does not hold, as the path $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_2 \rightarrow \dots$ violates the path property $FEGa$.

However, the formula $AGEXEGa$ holds.

3.2.3 CTL*

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

Definition 14 (Syntax of CTL*).

CTL* formulae can be built according to the following abstract syntax. CTL* state formulae over a set AP of atomic propositions can be built according to the abstract grammar

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

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

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

where ψ are CTL* state formulae. CTL* formulae are CTL* state formulae building the set $Form_{CTL^*}$.

$Form_{CTL^*}$

Again, we omit parentheses when it causes no confusion. We can define the “finally” and “globally” operators also for CTL* as syntactic sugar (see Exercise 6). 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 15 (Semantics of CTL*).

\models_{CTL^*}

The satisfaction relation $\models_{CTL^*} \subseteq \Sigma \times Form_{CTL^*}$ (or short \models) evaluates CTL* state

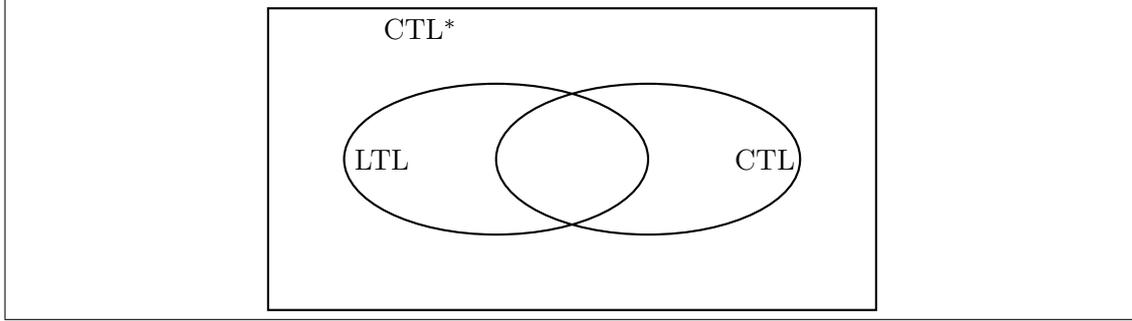


Figure 3.1: The expressiveness of LTL, CTL, and CTL*

formulae in the context of a state and CTL* path formulae in the context of a path as follows:

$$\begin{array}{ll}
 \sigma \models_{CTL^*} a & \text{iff } a \in L(\sigma) \\
 \sigma \models_{CTL^*} \psi_1 \wedge \psi_2 & \text{iff } \sigma \models_{CTL^*} \psi_1 \text{ and } \sigma \models_{CTL^*} \psi_2 \\
 \sigma \models_{CTL^*} \neg\psi & \text{iff } \sigma \not\models_{CTL^*} \psi \\
 \sigma \models_{CTL^*} E\varphi & \text{iff } \pi \models_{CTL^*} \varphi \text{ for some } \pi = \sigma_0 \rightarrow \sigma_1 \rightarrow \dots \text{ with } \sigma_0 = \sigma \\
 \\
 \pi \models_{CTL^*} \psi & \text{iff } \pi(0) \models_{CTL^*} \psi \\
 \pi \models_{CTL^*} \varphi_1 \wedge \varphi_2 & \text{iff } \pi \models_{CTL^*} \varphi_1 \text{ and } \pi \models_{CTL^*} \varphi_2 \\
 \pi \models_{CTL^*} \neg\varphi & \text{iff } \pi \not\models_{CTL^*} \varphi \\
 \pi \models_{CTL^*} \mathcal{X}\varphi & \text{iff } \pi^1 \models_{CTL^*} \varphi \\
 \pi \models_{CTL^*} \varphi_1 \mathcal{U} \varphi_2 & \text{iff } \text{exists } 0 \leq j \text{ with } \pi^j \models_{CTL^*} \varphi_2 \text{ and } \pi^i \models_{CTL^*} \varphi_1 \text{ for all } 0 \leq i < j.
 \end{array}$$

3.2.4 The Relation of LTL, CTL, and CTL*

The logics LTL and CTL are incomparable, and both are included in CTL*, as shown in Figure 3.1. 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 19.

- The LTL formula $\mathcal{F}\mathcal{G}a$ is not expressible in CTL.
- The CTL formula $A\mathcal{F}A\mathcal{G}a$ 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 7).

Example 20.

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

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

3.3 Model Checking

Model checking of discrete systems is not the basic content of this lecture, thus here we restrict ourselves to the intuition behind explicit¹ model checking 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.

Given a state transition system (usually represented as a Kripke-structure) and a CTL formula ψ_0 , CTL model checking labels the states recursively with the subformulae of ψ_0 inside-out.

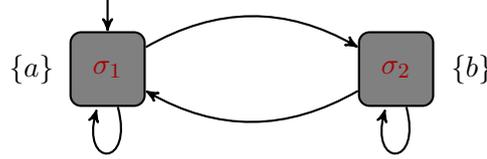
- The labeling with atomic propositions $a \in AP$ is given by a labeling function.
- Given the labelings for ψ_1 and ψ_2 , we label those states with $\psi_1 \wedge \psi_2$ that are labeled with both ψ_1 and ψ_2 .
- Given the labeling for ψ , we label those states with $\neg\psi$ that are not labeled with ψ .
- Given the labeling for ψ , we label those states with $E\mathcal{X}\psi$ that have a successor state labeled with ψ .
- 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.
- Given the labeling for ψ , we label those states with $A\mathcal{X}\psi$ whose successor states are all labeled with ψ .
- 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.

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. The formula ψ_0 is satisfied by the LSTS iff after termination of the procedure the initial state is labeled with ψ_0 . Note that this model checking approach would not be complete, i.e., in general would not terminate, for infinite-state systems.

¹Explicit model checking is based on the enumeration of states, in contrast to symbolic model checking using a symbolic BDD state space representation. CTL model checking.

Example 21.

Assume again the LSTS from Example 15:



In Example 18 we stated that this LSTS satisfies the CTL formula $AGEXEGa$. We prove this fact by model checking.

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

$$\begin{aligned} EG\psi &\leftrightarrow \neg Atrue \mathcal{U} \neg\psi \\ AG\psi &\leftrightarrow \neg Etrue \mathcal{U} \neg\psi. \end{aligned}$$

This yields

$$\psi := \neg (Etrue \mathcal{U} \neg (EX\neg (Atrue \mathcal{U} (\neg a))))$$

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

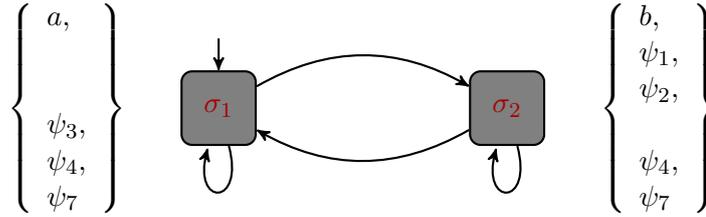
1. $\psi_1 := \neg a$
2. $\psi_2 := Atrue \mathcal{U} \psi_1$
3. $\psi_3 := \neg\psi_2$
4. $\psi_4 := EX\psi_3$
5. $\psi_5 := \neg\psi_4$
6. $\psi_6 := Etrue \mathcal{U} \psi_5$
7. $\psi_7 := \neg\psi_6$

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. $\psi_2 := Atrue \mathcal{U} \psi_1$:
 - 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. $\psi_4 := EX\psi_3$: 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. $\psi_6 := Etrue \mathcal{U} \psi_5$:
 - 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 .

3.4 Discrete-Time Systems

Though discrete systems have no continuous components in their model, the real-time behaviour of the modeled systems may nonetheless be relevant. Assume a controller executing a program. Though the program itself can be modeled as a discrete system, 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 behaviour 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 CTL and LTL also for their discrete-time extensions.

Note 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

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

Thus $\mathcal{X}^k\varphi = \underbrace{\mathcal{X} \dots \mathcal{X}}_k \varphi$ in LTL.

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

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

Thus $E\mathcal{X}^k\psi = \underbrace{E\mathcal{X} \dots E\mathcal{X}}_k \psi$. 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

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

In CTL we define

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

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

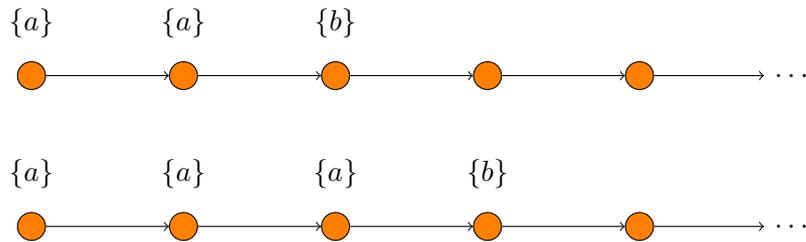
- \mathcal{U} for $\mathcal{U}^{[0,\infty]}$.

Example 22.

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].

Exercises

Exercise 5.

The LTL definition $\mathcal{G}\psi = \neg(\text{true } \mathcal{U} \neg\psi)$ of “globally” has to be modified for CTL since it does not follow the CTL syntax. Give a syntactically correct definition for the “globally” operator in CTL, i.e., define $\mathcal{G}\psi$ as a CTL path formula for a CTL state formula ψ .

Exercise 6.

Give a definition of the “finally” and “globally” operators in CTL* as syntactic sugar.

Exercise 7.

To show that CTL* is more expressive than LTL and CTL together, give a CTL* formula that can be expressed neither in LTL nor in CTL.

Chapter 4

Timed Automata

The general reachability problem for hybrid automata is undecidable [ACHH92]. However, there are subclasses of hybrid automata with restricted expressiveness that are decidable.

One of the most popular subclasses of hybrid automata is that of timed automata. Though their expressiveness is very restricted, a wide range of real-time systems can be modeled with timed automata, for which the reachability problem is decidable, and thus model checking can be applied. Basically, the only continuous component in timed automata is *time*.

In this chapter we introduce timed automata in Section 4.1. In Section 4.2 we extend the logic CTL with continuous time aspects, resulting in the logic 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 Timed Automata

Timed automata are a subclass of hybrid automata, putting several restrictions on the expressiveness. The variable set of a timed automaton is a finite set of *clocks*. A clock is a variable measuring the time, i.e., always evolving at rate 1. The values of the clocks can only be accessed in a limited way. For reading, 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 writing, clocks can only be *reset*, i.e., their values can only be set to 0.

Definition 16 (Syntax of clock constraints).

Clock constraints over a finite set \mathcal{C} of clocks can be built using the following abstract grammar:

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

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 $CC(\mathcal{C})$.

We also introduce clock constraints of the form $x = c$ as syntactic sugar, using the definition $x \geq c \wedge x \leq c$.

Clock constraints are evaluated in the context of a valuation $\nu \in V$ as introduced for hybrid automata. In the context of timed automata the valuation is a function assigning values to the clocks, i.e., being of type $\nu : \mathcal{C} \rightarrow \mathbb{R}$.

Definition 17 (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:

$$\begin{aligned} \nu \models_{CC} x < c & \quad \text{iff} \quad \nu(x) < c, \\ \nu \models_{CC} x \leq c & \quad \text{iff} \quad \nu(x) \leq c, \\ \nu \models_{CC} x > c & \quad \text{iff} \quad \nu(x) > c, \\ \nu \models_{CC} x \geq c & \quad \text{iff} \quad \nu(x) \geq c, \\ \nu \models_{CC} g_1 \wedge g_2 & \quad \text{iff} \quad \nu \models_{CC} g_1 \text{ and } \nu \models_{CC} g_2. \end{aligned}$$

We also use the notation $\sigma \models_{CC} g$ for states $\sigma = (l, \nu)$ with the meaning $\nu \models_{CC} g$.

For the sake of readability we also use notations like

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

with the expected meaning.

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

Definition 18 (Syntax of clock reset).

Given a finite set \mathcal{C} of clocks, a clock reset is an expression of the form $\text{reset}(C)$ with $C \subseteq \mathcal{C}$. Sometimes we also write $\text{reset}(x_1, \dots, x_n)$ for $\text{reset}(\{x_1, \dots, x_n\})$.

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

Definition 19 (Semantics of clock reset).

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

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

for all $x \in \mathcal{C}$. For a state $\sigma = (l, \nu)$ also write $\text{reset}(x)$ in σ for the state $(l, \text{reset}(x) \text{ in } \nu)$.

In the following we make use of the following notation for time delay:

Definition 20.

We define for all valuations $\nu \in V$ and constants $c \in \mathbb{N}$ the notation $\nu + c$ for the valuation with $(\nu + c)(x) = \nu(x) + c$ for all $x \in \mathcal{C}$.

 $\nu + c$
Example 23 (Clock access).

Assume a clock set $\mathcal{C} = \{x, y\}$ and a valuation $\nu : \mathcal{C} \rightarrow \mathbb{R}$ with $\nu(x) = 2$ and $\nu(y) = 3$. Then

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

Next we give the definition of timed automata. The definition of hybrid automata gets thereby restricted according to the above described properties as follows:

- The variables of a timed automaton are clocks. We write \mathcal{C} instead of Var to denote this fact.
- The activity functions can be skipped, as all clocks evolve with the derivative 1. That means, all timed automata with a clock set \mathcal{C} have the same set

$$\{f : \mathbb{R} \rightarrow V \mid \forall x \in \mathcal{C}. \exists c \in \mathbb{R}. \forall t \in \mathbb{R}_{\geq 0}. f(t)(x) = t + c\}$$

of activity functions in all locations, which we do not define explicitly.

- 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

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

For simplicity in the following we write edges in the form $(l, a, (g, C), l') \in Loc \times Lab \times (CC(\mathcal{C}) \times 2^{\mathcal{C}}) \times Loc$.

- The invariants are also of a restricted form, as they must be definable by clock constraints. An invariant set defined by a clock constraint $g \in CC(\mathcal{C})$ is given by the set of all valuations satisfying g :

$$\{\nu \in V \mid \nu \models g\}$$

Note that since all clock constraints are conjunctions of atomic clock constraints, the resulting invariant sets are convex. In the following we see the invariant function as of type $Inv : Loc \rightarrow CC(\mathcal{C})$ assigning a clock constraint to each location.

¹We can also allow $c \in \mathbb{Q}$.

- Finally, the valuation components of initial states assign the value 0 to all clocks.

There is a further difference between timed automata and hybrid automata regarding the parallel composition. For hybrid automata the parallel composition requires a common variable set, i.e., there are variables shared by the components. However, allowing shared variables in the timed automata composition leads to some complications, which we do not discuss here. Instead, we restrict the composition of timed automata such that the components must have disjoint variable sets, and give a definition of the parallel composition for timed automata under this restriction.

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 21 (Syntax of timed automata).

A timed automaton $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ is a tuple with

- Loc is a finite set of locations,
- \mathcal{C} is a finite set of 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,
- $Inv : Loc \rightarrow CC(\mathcal{C})$ is a function assigning an invariant to each location, and
- $Init \subseteq \Sigma$ with $\nu(x) = 0$ for all $x \in \mathcal{C}$ and all $(l, \nu) \in Init$.

We call the variables in \mathcal{C} clocks. We also use the notation² $l \xrightarrow{a:g,C} l'$ to state that there exists an edge $(l, a, (g, C), l') \in Edge$.

Definition 22 (Semantics of timed automata).

The semantics of a timed automaton $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ is given by an operational semantics, consisting of the following two rules:

$$\frac{(l, a, (g, C), l') \in Edge \quad \nu \models g \quad \nu' = \text{reset}(C) \text{ in } \nu \quad \nu' \models Inv(l')}{(l, \nu) \xrightarrow{a} (l', \nu')} \text{Rule}_{Discrete}$$

$$\frac{t \in \mathbb{R}_{\geq 0} \quad \nu' = \nu + t \quad \nu' \models Inv(l)}{(l, \nu) \xrightarrow{t} (l, \nu')} \text{Rule}_{Time} .$$

We use \rightarrow to denote $\xrightarrow{a} \cup \xrightarrow{t}$ when we are not interested in the type of the steps.

²Note that we deviate from the notation in the book of Baier and Katoen who use $l \xrightarrow{g:a,C} l'$.

A run (or path or execution) of \mathcal{T} is an infinite sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \dots$ with $\sigma_i \in \Sigma$, $\sigma_0 = (l_0, \nu_0) \in \text{Init} \cap \text{Inv}(l_0)$. We use $\text{Paths}^{\mathcal{T}}$ (or short *Paths*) to denote the set of all paths of \mathcal{T} and $\text{Reach}^{\mathcal{T}}$ (or short *Reach*) for the set of reachable states of \mathcal{T} , i.e., the set of states for that there is a path containing that state. We write $\text{Paths}(\sigma_0)$ for the set of infinite execution fragments $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \dots$ starting in a not necessarily initial state σ_0 .

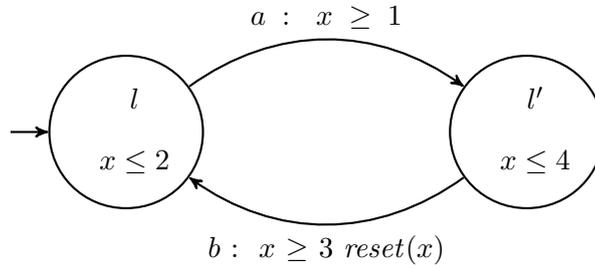
Note that, since the invariants are convex, it is enough to require that they hold *after* each time step, and we do not need the requirement that they hold *during* the whole period of a time step. Together with the fact that paths start in an initial state satisfying the invariant of the initial location, 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 $L : \text{Loc} \rightarrow 2^{AP}$ where AP denotes the set of atomic propositions. To simplify the notations, we overload the labeling function defining $L : \Sigma \rightarrow 2^{AP}$ with $L((l, \nu)) = L(l)$.

Similarly to hybrid automata, also timed automata is often represented graphically. As all clocks evolve with derivative 1 we do not represent the time behaviour in the graphs. Trivial conditions, non-synchronizing labels, and empty reset sets are also skipped.

Example 24.

The graphical representation



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

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

Definition 23 (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 automaton 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 $\nu : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$, valuation set V , and states $\Sigma = Loc \times V$, where

- $Loc = Loc_1 \times Loc_2$,
- $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$,
- $Lab = Lab_1 \cup Lab_2$,
- $Inv((l_1, l_2)) = Inv_1(l_1) \wedge Inv_2(l_2)$ for all $(l_1, l_2) \in Loc$,
- $Init = \{((l_1, l_2), \nu) \in \Sigma \mid (l_1, \nu) \in Init_1 \wedge (l_2, \nu) \in Init_2\}$.

The discrete transitions in $Edge$ are defined by the following rules:

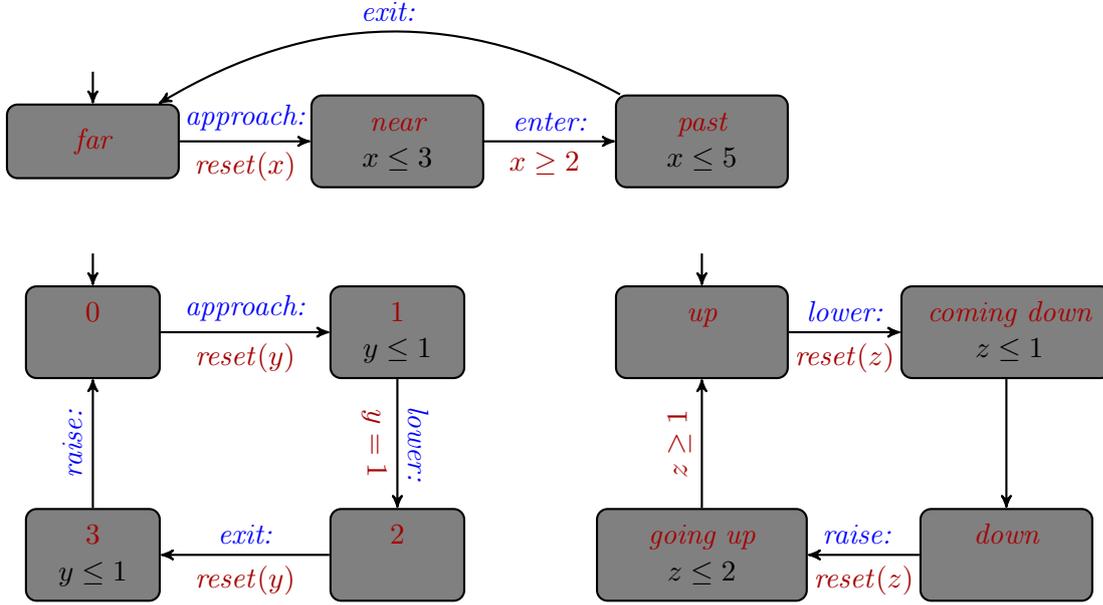
$$\frac{(l_1, a, (g_1, \mathcal{C}_1), l'_1) \in Edge_1 \quad (l_2, a, (g_2, \mathcal{C}_2), l'_2) \in Edge_2}{((l_1, l_2), a, (g_1 \wedge g_2, \mathcal{C}_1 \cup \mathcal{C}_2), (l'_1, l'_2)) \in Edge} \quad \text{Rule}_{Synch}$$

$$\frac{(l_1, a, (g, \mathcal{C}), l'_1) \in Edge_1 \quad a \notin Lab_2}{((l_1, l_2), a, (g, \mathcal{C}), (l'_1, l_2)) \in Edge} \quad \text{Rule}_{NonSynch_1}$$

$$\frac{(l_2, a, (g, \mathcal{C}), l'_2) \in Edge_2 \quad a \notin Lab_1}{((l_1, l_2), a, (g, \mathcal{C}), (l_1, l'_2)) \in Edge} \quad \text{Rule}_{NonSynch_2}$$

Example 25.

The railroad crossing Example 14 is modeled as a parallel composition of timed automata. Adapting the syntax of timed automata we get the following graphical representation:



4.1.1 Continuous-Time Phenomena

In the state space model of timed automata induced by their semantics, each path corresponds to a possible system behaviour. However, some of the paths may model unrealistic behaviour.

Time convergence: There are syntactically unavoidable paths of timed automata along which time converges, i.e., time never evolves beyond some value. For example, the timed automaton from Example 24 has a path

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

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 $n \rightarrow \infty$. 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 for the property specification.

Time lock: 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 *time locks*. Timed automata without time locks are called *time-lock free*. Time locks are modeling flaws, i.e., they can be avoided by appropriate modeling.

Zenoness: Paths on 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. Zeno paths are also modeling flaws, and they can be avoided by careful modeling.

Next we formalize the above properties.

Definition 24 (Time convergence, time lock, zenoness).

Let $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ be a timed automaton. We define the time duration of a step by the function $ExecTime : (Lab \cup \mathbb{R}_{\geq 0}) \rightarrow \mathbb{R}_{\geq 0}$ with

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

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

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

- An infinite execution fragment π is said to be time-divergent if $ExecTime(\pi) = \infty$, and time-convergent otherwise.

$Paths_{div}$

For a state $\sigma \in \Sigma$ we define $Paths_{div}(\sigma) \subseteq Paths(\sigma)$ to be the set of time-divergent infinite execution fragments starting in σ .

- A state $\sigma \in \Sigma$ contains a timelock iff $Paths_{div}(\sigma) = \emptyset$. A timed automaton is said to be timelock-free if none of its reachable states contains a timelock.
- π is said to be zeno if it is time-convergent and infinitely many discrete actions are executed within π . \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 for non-zenoness, which is simple to check.

Theorem 1 (Sufficient condition for non-zenoness).

Let $\mathcal{T} = (Loc, \mathcal{C}, Lab, Edge, Inv, Init)$ be a timed automaton such that for each sequence of edges

$$l_0 \xrightarrow{\alpha_1: g_1, C_1} l_1 \xrightarrow{\alpha_2: g_2, C_2} l_2 \dots \xrightarrow{\alpha_n: g_n, C_n} l_n$$

with $l_0 = l_n$ 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

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

for some $0 < j \leq n$.

Then \mathcal{T} is non-zeno.

4.2 Timed Temporal Logics

Hybrid automata in general and timed automata in special usually model real-time systems, which are often *time-critical* in the sense that for their correct functioning certain events must occur within some time limits. 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 those 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.

TCTL extends CTL, with the following main differences:

- For discrete systems we used an atomic proposition set and a labeling function to assign atomic propositions to states. To be able to argue about the validity of the atomic propositions in CTL, they are defined to be atomic CTL state formulae.
Besides such atomic propositions, for timed automata we also need to be able to argue about the clock values in form of atomic clock constraints. Thus atomic clock constraints are also defined to be 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, TCTL quantification ranges over time-divergent paths, only.

The syntax of TCTL is as follows:

Definition 25 (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

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

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

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

with $J \subseteq \mathbb{R}^{\geq 0}$ is an 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.

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

Definition 26 (Semantics of TCTL).

Let $\mathcal{T} = (\text{Loc}, \mathcal{C}, \text{Lab}, \text{Edge}, \text{Inv}, \text{Init})$ be a timed automaton, AP a set of atomic propositions, and $L : \text{Loc} \rightarrow 2^{AP \cup \text{ACC}(\mathcal{C})}$ a state labeling function. The function \models_{TCTL} (or short \models) assigns a truth value to each TCTL state and path formulae as follows:

$$\begin{aligned} \sigma &\models_{\text{TCTL}} \text{true} \\ \sigma &\models_{\text{TCTL}} a \quad \text{iff } a \in L(\sigma) \\ \sigma &\models_{\text{TCTL}} g \quad \text{iff } \sigma \models_{\text{CC}} g \\ \sigma &\models_{\text{TCTL}} \neg \psi \quad \text{iff } \sigma \not\models_{\text{TCTL}} \psi \\ \sigma &\models_{\text{TCTL}} \psi_1 \wedge \psi_2 \quad \text{iff } \sigma \models_{\text{TCTL}} \psi_1 \text{ and } \sigma \models_{\text{TCTL}} \psi_2 \\ \sigma &\models_{\text{TCTL}} E\varphi \quad \text{iff } \pi \models_{\text{TCTL}} \varphi \text{ for some } \pi \in \text{Paths}_{\text{div}}(\sigma) \\ \sigma &\models_{\text{TCTL}} A\varphi \quad \text{iff } \pi \models_{\text{TCTL}} \varphi \text{ for all } \pi \in \text{Paths}_{\text{div}}(\sigma). \end{aligned}$$

where $\sigma \in \Sigma$, $a \in AP$, $g \in \text{ACC}(\mathcal{C})$, $J \subseteq \mathbb{R}_{\geq 0}$ a real-valued interval, ψ , ψ_1 and ψ_2 are TCTL state formulae, and φ is a TCTL path formula.

For an infinite execution fragement $\pi = \sigma_0 \xrightarrow{\alpha_0} \sigma_1 \xrightarrow{\alpha_1} \sigma_2 \xrightarrow{\alpha_2} \dots \in \text{Paths}_{\text{div}}(\sigma_0)$ for some $\sigma_0 \in \Sigma$, the satisfaction relation for bounded until formulae is defined by

$$\begin{aligned} \pi \models_{\text{TCTL}} \psi_1 \mathcal{U}^J \psi_2 \quad \text{iff} \quad &\exists i \geq 0. \sigma_i + d \models_{\text{TCTL}} \psi_2 \text{ for some } d \in [0, d_i] \\ &\text{with } (\sum_{k=0}^{i-1} d_k) + d \in J \\ &\text{and } \forall j \leq i. \sigma_j + d' \models_{\text{TCTL}} \psi_1 \vee \psi_2 \text{ for any } d' \in [0, d_j] \text{ with} \\ &\text{either } j < i \text{ or } d' \leq d. \end{aligned}$$

Sat

where $d_i = \text{ExecTime}(\alpha_i)$. We define

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

and

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

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

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 if 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-zero 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 if $\mathcal{T} \models_{TCTL} \psi$

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 $RTA \models_{CTL} \hat{\psi}$.
4. Apply CTL model checking to check whether $RTA \models_{CTL} \hat{\psi}$.
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 $L : Loc \rightarrow 2^{AP}$, 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

$$\begin{aligned} \sigma \models_{TCTL} E(\psi_1 \mathcal{U}^J \psi_2) & \text{ iff } \text{reset}(z) \text{ in } \sigma \models_{TCTL} E((\psi_1 \vee \psi_2) \mathcal{U}((z \in J) \wedge \psi_2)) \\ \sigma \models_{TCTL} A(\psi_1 \mathcal{U}^J \psi_2) & \text{ iff } \text{reset}(z) \text{ in } \sigma \models_{TCTL} A((\psi_1 \vee \psi_2) \mathcal{U}((z \in J) \wedge \psi_2)) \end{aligned}$$

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. How we can use this transformation for nested formulae

will be explained later when we introduce the slightly adapted CTL model checking algorithm.

Example 26.

The TCTL formula $EF^{\leq 2}AG^{[2,3]}a$ gets transformed into $EF(z \leq 2 \wedge AG(2 \leq z \leq 3 \rightarrow a))$.

4.3.2 Finite State Space Abstraction

Since the state space of a timed automaton is in general infinite, in order to do model checking we want to define a *finite abstraction*. An abstraction represents 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 behaviour. 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 model checking is not possible, abstraction may give us a finite-state system for which we can do model checking.

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 , and a labeling function $L : \Sigma \rightarrow 2^{AP}$ assigning subsets of AP to the states of \mathcal{LSTS} . Assume furthermore 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 , then 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 behaviour 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 , then 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 behaviour 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 $\sigma_1 = (l_1, \nu_1) \in \Sigma$ and $\sigma_2 = (l_2, \nu_2) \in \Sigma$ 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., $L(l_1) = L(l_2)$, 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 , then 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 below, 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 $L : Loc \rightarrow 2^{AP}$, 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
- $frac(r)$ 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:

$$\nu \models x \leq c \Leftrightarrow \nu(x) \leq c \Leftrightarrow \lfloor \nu(x) \rfloor < c \vee (\lfloor \nu(x) \rfloor = c \wedge frac(\nu(x)) = 0).$$

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

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

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 $(l, \nu) \cong (l, \nu')$ should satisfy

$$(4.1) \quad (\nu(x) > c_x \wedge \nu'(x) > c_x) \quad \vee \\ (\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor \wedge frac(\nu(x)) = 0 \text{ iff } frac(\nu'(x)) = 0)$$

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

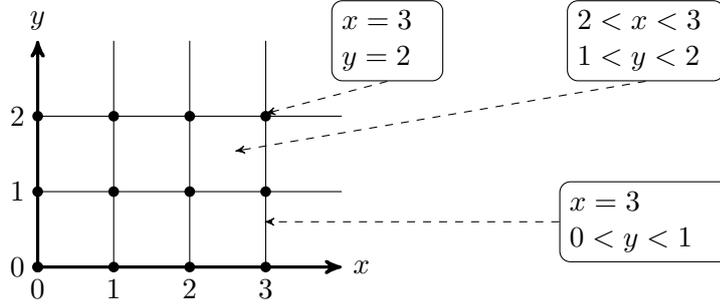
Example 27.

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 behaviour 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.

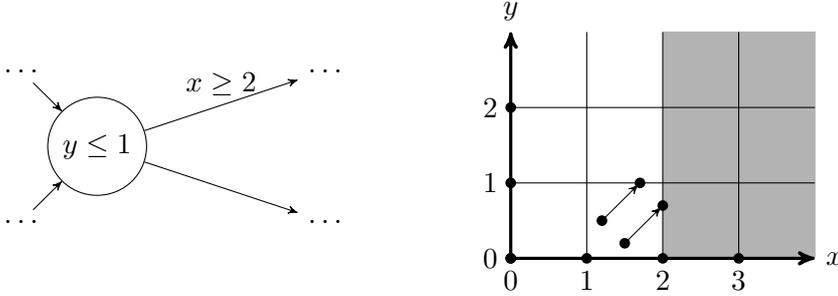
In the graphical representation below, valuations belonging to different points, line fragments, or boxes must not be equivalent. This yields at least 48 equivalence classes.



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

Example 28.

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



If control is in the location 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 behaviour.

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 $\nu(x), \nu'(x) \leq c_x \wedge \nu(y), \nu'(y) \leq c_y$

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

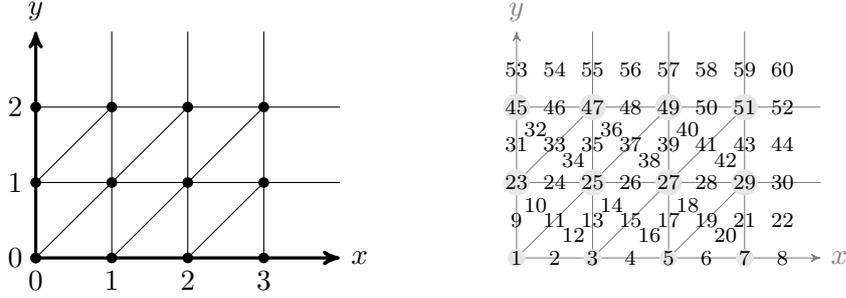
Because of symmetry requiring

$$\text{frac}(\nu(x)) \leq \text{frac}(\nu(y)) \quad \text{iff} \quad \text{frac}(\nu'(x)) \leq \text{frac}(\nu'(y)).$$

is sufficient.

Example 29.

We extend the graphical representation of the clock equivalence classes from Example 27 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 27.

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 $(l, \nu) \cong (l', \nu')$ iff $l = l'$ and

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

$$\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor \wedge (\text{frac}(\nu(x)) = 0 \text{ iff } \text{frac}(\nu'(x)) = 0)$$

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

$$\text{frac}(\nu(x)) \leq \text{frac}(\nu(y)) \text{ iff } \text{frac}(\nu'(x)) \leq \text{frac}(\nu'(y)).$$

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 $(l, \nu) \in \Sigma$ is the set $[(l, \nu)] = \{(l, \nu') \in \Sigma \mid \nu \cong \nu'\}$. We also write (l, r) for $\{(l, \nu) \mid \nu \in r\}$.

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

$$\begin{aligned} r \models g & \text{ iff } \forall \nu \in r. \nu \models g \\ (l, r) \models g & \text{ iff } r \models g. \end{aligned}$$

for r being a clock region of \mathcal{T} with clocks \mathcal{C} and a TCTL formula ψ , and $g \in \text{ACC}(\mathcal{T}) \cup \text{ACC}(\psi)$. 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:

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

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

Definition 28.

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 $r' = \text{succ}(r)$, 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 $\text{succ}((l, r)) = (l, \text{succ}(r))$.

Definition 29.

Let $\mathcal{T} = (\text{Loc}, \mathcal{C}, \text{Lab}, \text{Edge}, \text{Inv}, \text{Init})$ 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 labelled state transition system $\mathcal{RTS}(\mathcal{T}, \hat{\psi}) = (\Sigma', \text{Lab}', \text{Edge}', \text{Init}')$ with

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

and

$$\frac{(l, a, (g, \mathcal{C}), l') \in \text{Edge} \quad r \models g \quad r' = \text{reset}(\mathcal{C}) \text{ in } r \quad r' \models \text{Inv}(l')}{(l, [\nu]) \xrightarrow{a} (l', [\nu'])} \quad \text{Rule}_{\text{Discrete}}$$

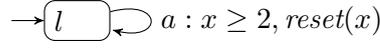
$$\frac{r \models \text{Inv}(l) \quad \text{succ}(r) \models \text{Inv}(l)}{(l, r) \xrightarrow{\tau} (l, \text{succ}(r))} \quad \text{Rule}_{\text{Time}}$$

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

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

Example 30.

Assume the following timed automaton having a single clock x :



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

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

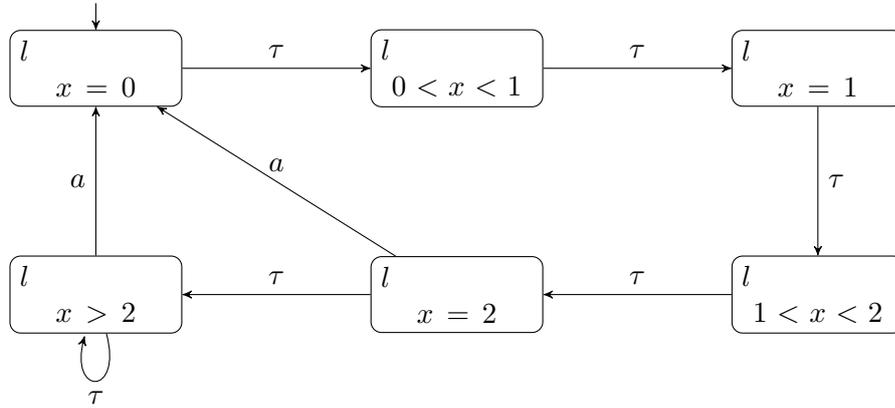
For the transitions, τ -transitions are defined from each region into its successor region:

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

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

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

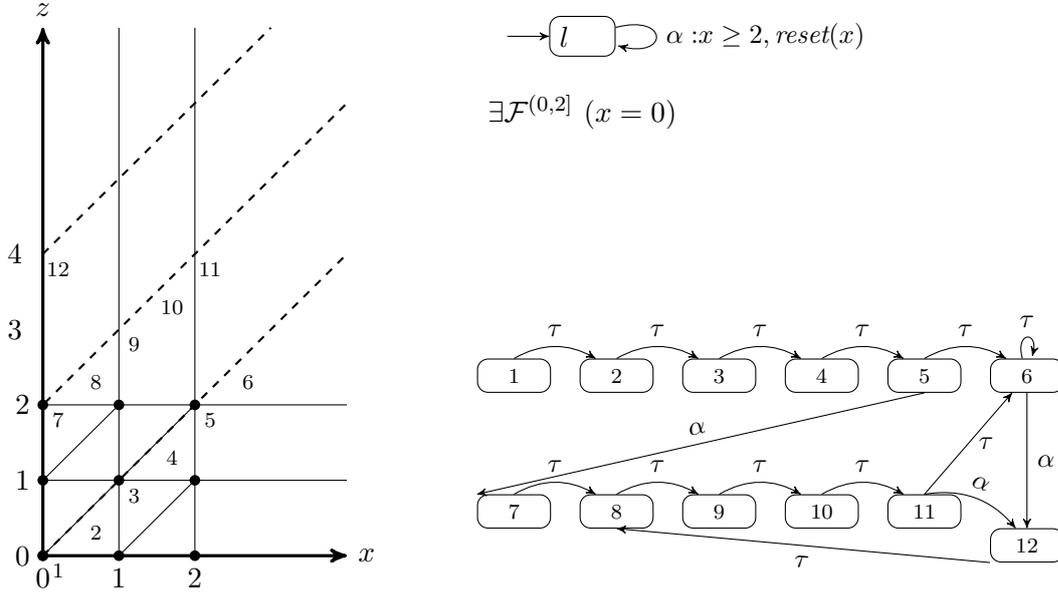
The resulting region transition graph can be visualized as follows, where for clarity we write into the states the locations and the constraints to which they correspond:



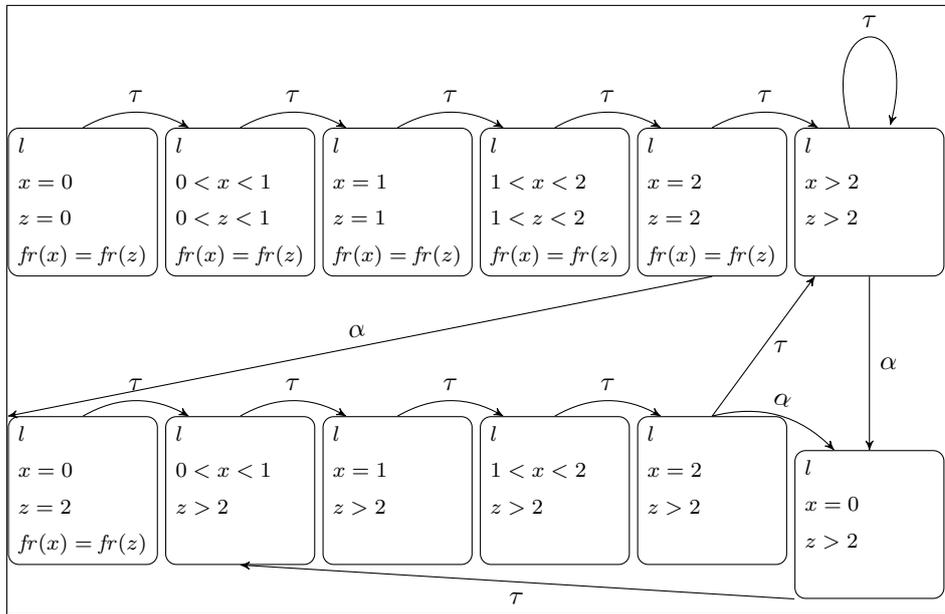
Example 31.

Assume the same timed automaton as in the previous Example 30, but now additionally consider the TCTL formula $EF^{(0,2]}(x = 0)$. After removing the bound we get the unbounded formula $EF(0 < z \leq 2 \wedge x = 0)$. 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 behaviours, 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 1.

For non-zeno \mathcal{T} and $\pi = s_0 \rightarrow s_1 \rightarrow \dots$ 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 $s_i \in (l, r)$ for all $i \geq j$.
- if there is a state region (l, r) with $r \neq r_\infty$ and an index j such that $s_i \in (l, r)$ for all $i \geq j$ then π is time-convergent.

Theorem 2.

A non-zero 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 2.

For a non-zero timed automaton \mathcal{T} and an unbounded TCTL formula ψ :

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

Lemma 3.

The model checking problem for timed automata and TCTL is complete for PSPACE.

Exercises

Exercise 8.

Prove Theorem 2.

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 of Rectangular Automata

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 $Var = \{x_1, \dots, x_d\}$ of the automata are ordered and write $(l, \nu) \in Loc \times \mathbb{R}^d$ for a state (l, ν) with $\nu(x_i) = v_i$ for all $i = 1, \dots, d$.

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

Definition 30 (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 31 (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 $Var = \{x_1, \dots, x_d\}$ of d ordered real-valued variables; we write $x = (x_1, \dots, x_d)$ for the ordered sequence of the variables;
- a function $Con : Loc \rightarrow 2^{Var}$ assigning a set of controlled variables to each location;
- a finite set Lab of synchronization labels;
- a set $Edge \subseteq Loc \times Lab \times (\mathcal{R}^d \times \mathcal{R}^d \times 2^{\{1, \dots, n\}}) \times Loc$ of edges;
- a flow function $Act : Loc \rightarrow \mathcal{R}^d$;
- an invariant function $Inv : Loc \rightarrow \mathcal{R}^d$;
- initial states $Init : Loc \rightarrow \mathcal{R}^d$.

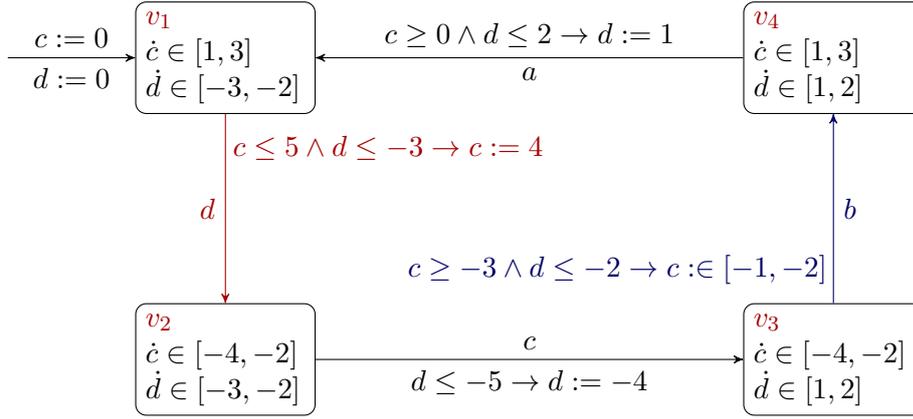
A rectangular automaton is initialized iff for all edges $e = (l, a, pre, post, jump, l') \in Edge$ and all $i \in \{1, \dots, n\}$ we have that $Act(l)_i \neq Act(l')_i$ implies $i \in jump$, where $Act(l)_i$ is the projection of $Act(l)$ to the i th dimension.

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 i th 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 32.

The following hybrid automaton 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 32 (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:

$$\frac{(l, a, pre, post, jump, l') \in Edge \quad v \in pre \quad v' \in post \quad \forall i \notin jump. v'_i = v_i \quad v' \in Inv(l')}{(l, v) \xrightarrow{a} (l', v')} \quad \text{Rule}_{Discrete}$$

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

The one-step transition is given by $\rightarrow = \xrightarrow{a} \cup \xrightarrow{t}$, its transitive closure by \rightarrow^* . A path is a sequence $\sigma_0 \rightarrow \sigma_1 \rightarrow \sigma_2 \dots$ starting in an initial state $\sigma_0 = (l_0, v_0)$ with $v_0 \in Init(l_0) \cap Inv(l_0)$. 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 4.

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.

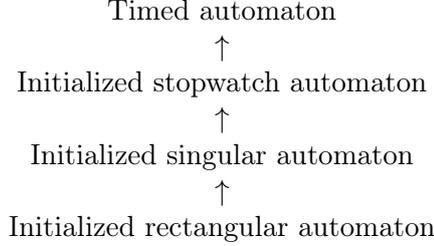
5.2 Decidability of Rectangular Automata

The reachability problem for initialized rectangular automata is decidable.

Lemma 5.

The reachability problem for initialized rectangular automata is complete for PSPACE.

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



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 33.

- 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 6.

The reachability problem for initialized stopwatch automata is complete for PSPACE.

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} is a d -dimensional initialized stopwatch automaton with location set $Loc_{\mathcal{H}}$. Let κ be the set of rational constants used in the definition of \mathcal{H} , and let $\kappa_{\perp} = \kappa \cup \{\perp\}$.

We define a d -dimensional timed automaton \mathcal{H}' with locations $Loc_{\mathcal{H}'} = Loc_{\mathcal{H}} \times \kappa_{\perp}^{\{1, \dots, d\}}$. Each location (l, f) of \mathcal{H}' consists of a location l of \mathcal{H} and a function $f : \{1, \dots, d\} \rightarrow \kappa_{\perp}$.

Each state $q = ((l, f), v)$ of \mathcal{H}' represents the state $\alpha(q) = (l, u)$ of \mathcal{H} , where $u_i = v_i$ if $f(i) = \perp$, and $u_i = f(i)$ if $f(i) \neq \perp$.

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

5.2.2 From Initialized Singular Automata to Initialized Stopwatch Automata

Definition 34.

- A variable x_i is a finite-slope variable if $\text{flow}(l)_i$ 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 7.

The reachability problem for initialized singular automata is complete for PSPACE.

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 $\beta : \mathbb{R}^d \rightarrow \mathbb{R}^d$ defined as follows:

For each location l of B , if $\text{Act}_B(l) = \prod_{i=1}^d [k_i, k_i]$, then $\beta(v_1, \dots, v_d) = (l_1 \cdot v_1, \dots, l_d \cdot v_d)$ with $l_i = k_i$ if $k_i \neq 0$, and $l_i = 1$ 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 .

We have:

- The reachable set of $\text{Reach}(B)$ of B is $\beta(\text{Reach}(C_B))$.
- $\text{Lang}(B) = \text{Lang}(C_B)$

5.2.3 From Initialized Rectangular Automaton to Initialized Singular Automaton

Lemma 8.

The reachability problem for initialized rectangular automata is complete for PSPACE.

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

In this chapter we discuss a further class of hybrid automata called *linear hybrid automata*. Linear hybrid automata 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 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.

We introduce linear hybrid automata 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 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 35.

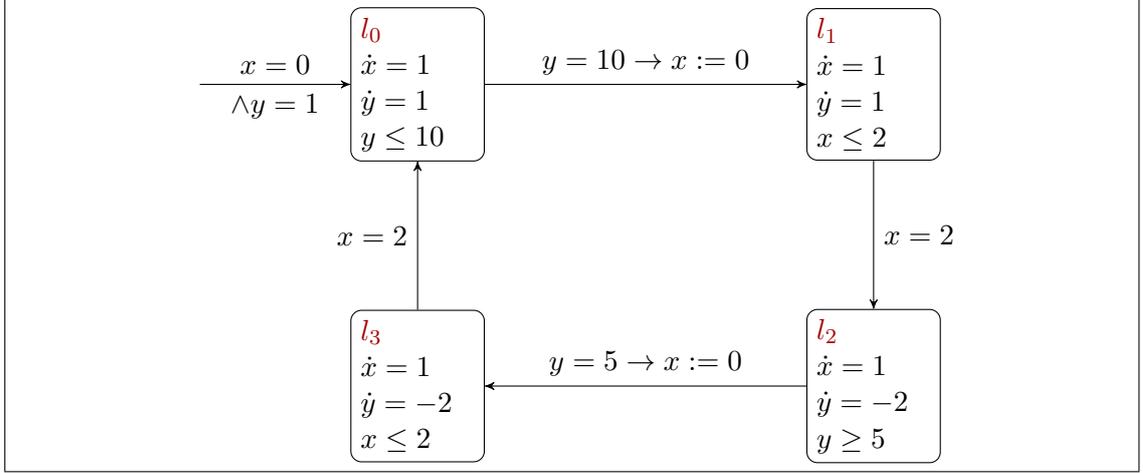
- A linear term over the set Var of variables is a linear combination of variables in Var with integer (rational) coefficients.
- A linear formula over Var is a Boolean combination of (in)equalities between linear terms over Var .
- A hybrid automaton is time deterministic iff for every location $l \in Loc$ 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 $f_l^x[\nu]$.

$f_l[\nu]$
 $f_l^x[\nu]$

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

Definition 36 (Syntax of linear hybrid automata).

A linear hybrid automaton is a time-deterministic hybrid automaton with the following properties:


Figure 6.1: Water-level monitor

- Activities $Act(l)$ are given as sets of differential equations $\dot{x} = k_x$, one for each variable $x \in Var$, with k_x an integer (rational) constant:

$$f_l^x[\nu](t) = \nu(x) + k_x \cdot t.$$

- Invariants $Inv(l)$ are defined by linear formulae ψ over Var :

$$\nu \in Inv(l) \quad \text{iff} \quad \nu \models \psi$$

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

$$\psi \Rightarrow \{x := [\alpha_x, \beta_x] \mid x \in Var\},$$

where the guard ψ is a linear formula and α_x, β_x are linear terms. For the transition relation $\mu \subseteq V^2$ we have

$$(\nu, \nu') \in \mu \quad \text{iff} \quad \nu \models \psi \wedge \forall x \in Var. \nu(\alpha_x) \leq \nu'(x) \leq \nu(\beta_x).$$

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

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

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

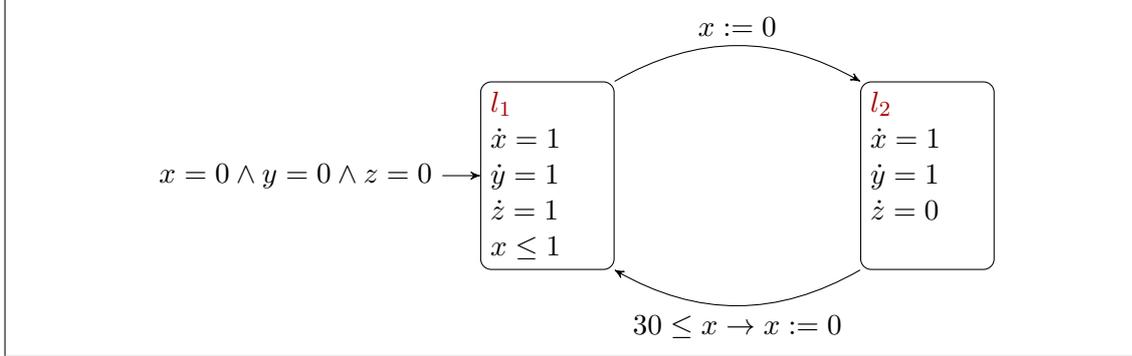


Figure 6.2: Leaking gas burner

$$\begin{array}{c}
 f \in \text{Act}(l) \quad f(0) = \nu \quad f(t) = \nu' \\
 t \geq 0 \quad \forall 0 \leq t' \leq t. f(t') \in \text{Inv}(l) \\
 \hline
 \text{Rule}_{\text{Time}} \\
 (l, \nu) \xrightarrow{t} (l, \nu')
 \end{array}$$

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

Definition 37.

For time-deterministic hybrid automata we define the “time can progress” predicate:

 $tcp_l[\nu](t)$

$$tcp_l[\nu](t) \quad \text{iff} \quad \forall 0 \leq t' \leq t. f_l[\nu](t') \in \text{Inv}(l).$$

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

$$\begin{array}{c}
 t \geq 0 \quad tcp_l[\nu](t) \\
 \hline
 \text{Rule}'_{\text{Time}} \\
 (l, \nu) \xrightarrow{t} (l, f_l[\nu](t))
 \end{array}$$

6.2 Forward analysis

The reachability problem for linear hybrid automata is in general undecidable. However, bounded reachability is still decidable. Despite of undecidability, for the general reachability analysis of linear hybrid automata 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:

Definition 38.

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

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

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

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

For the discrete steps, the corresponding one-step relation is formalized by postconditions: We define the postcondition $\text{post}_e[P]$ of P with respect to an edge $e = (l, a, \mu, l')$ as the set of valuations reachable from P by e :

$\text{post}_e[P]$

$$\nu' \in \text{post}_e[P] \quad \text{iff} \quad \exists \nu \in P. (\nu, \nu') \in \mu.$$

$\text{post}[R]$

An extension to regions $R = \bigcup_{l \in \text{Loc}} (l, R_l)$ is defined as follows:

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

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

Lemma 9.

For all linear hybrid automata, if $P \subseteq V$ is a linear set of valuations, then for all $l \in \text{Loc}$ and $e \in \text{Edge}$, both $\langle P \rangle_l^\nearrow$ and $\text{post}_e[P]$ 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.

Definition 39.

(I, \mapsto^*)

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 :

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

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 10.

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

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

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

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

6.3 Backward analysis

There is a similar backward approach for the fixed-point-based reachability analysis of linear hybrid automata. 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 40.

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:

$$\nu' \in \langle P \rangle_l^\swarrow \quad \text{iff} \quad \exists \nu \in P. \exists t \in \mathbb{R}^{\geq 0}. \text{tcp}_l[\nu'](t) \wedge \nu = \text{ft}[\nu'](t).$$

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

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

We define the precondition $\text{pre}_e[P]$ 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 :

$$\nu' \in \text{pre}_e[P] \quad \text{iff} \quad \exists \nu \in P. (\nu', \nu) \in \mu.$$

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

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

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

Lemma 11.

For all linear hybrid automata, 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 41.

$(\mapsto^* R)$ 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:

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

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

Lemma 12.

Let $R = \cup_{l \in Loc} (l, R_l)$ be a region of the linear hybrid automaton 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

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

6.4 Approximate 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 6.3). 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 it holds also for the over-approximated reachable set. On the other hand, if the proof fails, then, due

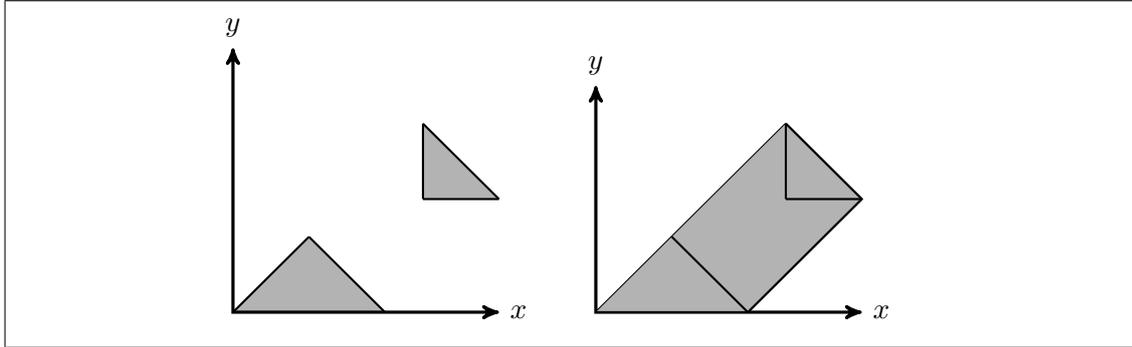


Figure 6.3: Two sets (left) and their convex hull (right)

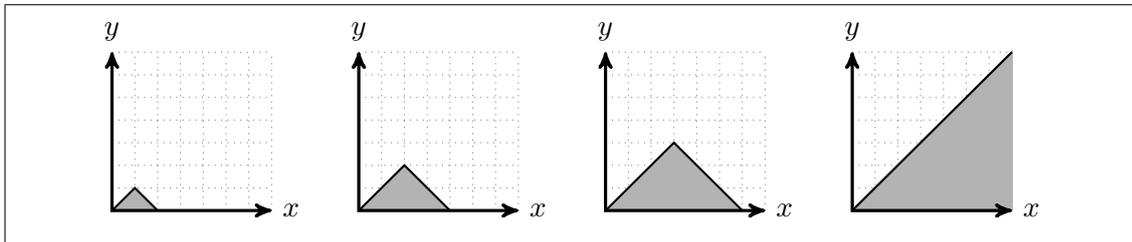


Figure 6.4: A sequence of three sets (left three pictures) and their widening (right)

to the over-approximation, it does not mean that the property does not hold: those states of the over-approximation that violate the property 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 6.4 illustrates the widening technique.

6.5 Minimization

In this section we discuss another approach called *minimization* for the analysis of linear hybrid automata, 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 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.

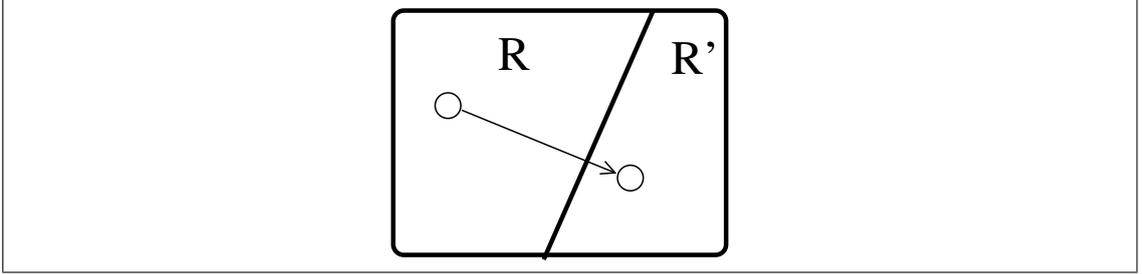


Figure 6.5: The next relation \mapsto on regions

The abstraction is based on partitioning the state space of a linear hybrid automaton into a finite set $\Pi = \{R_{bad}, R_1, \dots, R_n\}$ 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. 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 we are only interested in the reachability of bad states, we define no successors for R_{bad} . The abstract transitions are formalized as follows:

Definition 42.

\mapsto The next relation \mapsto on regions is defined by

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

Figure 6.5 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 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

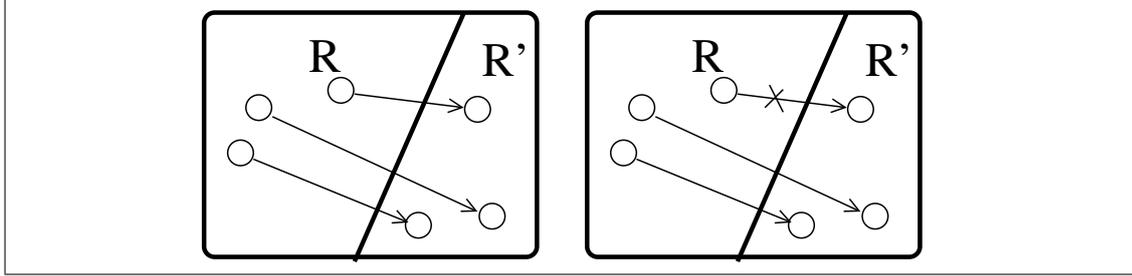


Figure 6.6: Stability of regions: a stable region (left) and a non-stable one (right)

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 $\sigma \rightarrow \sigma'$ we can conclude that there is at least one reachable state in R' .

Definition 43.

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

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

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 6.6 illustrates the stability of regions.

Now we come to the algorithm as specified by Figure 6.7. 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 partitioning: the region R_{bad} contains all “bad” states and the region $\Sigma \setminus R_{bad}$ 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 initial 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.

```

minimize( $\Sigma, R_{bad}$ ) {
   $\Pi := \{R_{bad}, \Sigma \setminus R_{bad}\}$ ; reach :=  $\{R \in \Pi \mid R \cap I \neq \emptyset\}$ ; completed :=  $\emptyset$ ;
  while ( $R_{bad} \notin \textit{reach} \wedge \textit{reach} \neq \textit{completed}$ ) {
    choose  $R \in (\textit{reach} \setminus \textit{completed})$ ;  $S := \emptyset$ ;
    for each ( $R' \in (\Pi \setminus \textit{reach})$  with  $R \mapsto R'$ ) {
      reach' := split( $\Pi, R, R'$ );
      if (reach' =  $R$ ) then reach := reach  $\cup \{R'\}$ ;
      else  $S := \textit{reach}'$ ;
    }
    if ( $S = \emptyset$ ) then completed := completed  $\cup \{R\}$ ;
    else {
       $\Pi := (\Pi \setminus \{R\}) \cup S$ ;
      reach := (reach  $\setminus \{R\}$ )  $\cup \{S_i \in S \mid S_i \cap I \neq \emptyset\}$ ;
      completed := completed  $\setminus \{R' \in \Pi \mid \exists S_i \in (S \setminus \textit{reach}). R' \mapsto S_i\}$ ;
    }
  }
  return  $R_{bad} \in \textit{reach}$ ;
}
    
```

Figure 6.7: The minimization algorithm

Otherwise, if there is still a successor region $R' \notin \textit{reach}$ 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:

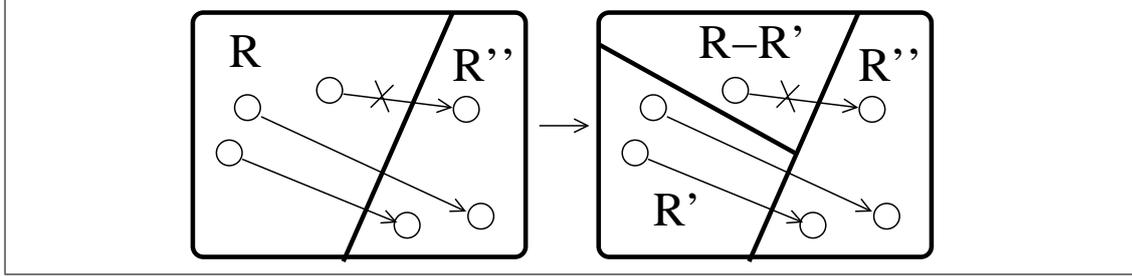
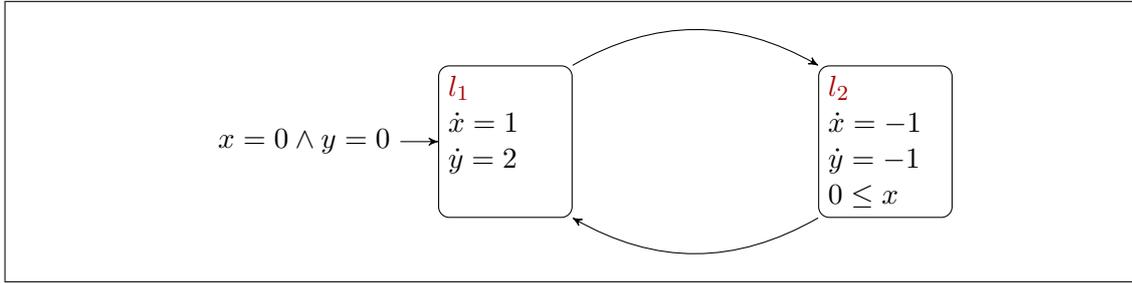
Definition 44.

split(Π, R, R')

$$\textit{split}(\Pi, R, R') := \begin{cases} \{R'', R \setminus R''\} & \text{if } R'' = \text{pre}[\langle R' \rangle^{\sphericalangle}] \cap R \wedge R'' \neq \emptyset \wedge R'' \neq R, \\ \{R\} & \text{otherwise.} \end{cases}$$

Figure 6.8 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


Figure 6.8: The splitting of regions

Figure 6.9: Leaking gas burner

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.

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 $R_{bad} \in reach$ then the system violates the property. Otherwise, if $R_{bad} \notin reach$ but $reach = completed$ 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 13.

The procedure in Figure 6.7 returns *TRUE* iff $I \mapsto^* R_{bad}$.

Example 33.

Assume the linear hybrid automaton shown in Figure 6.9. We want to prove that $0 \leq y$ always holds.

We have

$$\begin{aligned} R_{bad} &= (l_1, y < 0) \cup (l_2, 0 \leq x \wedge y < 0) \\ R_1 &= (l_1, 0 \leq y) \cup (l_2, 0 \leq x \wedge 0 \leq y) \end{aligned}$$

The algorithm initializes

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

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} :

$$\begin{aligned}\langle R_{bad} \rangle^{\leftarrow} &= \langle (l_1, y < 0) \cup (l_2, 0 \leq x \wedge y < 0) \rangle^{\leftarrow} \\ &= \langle (l_1, y < 0) \rangle^{\leftarrow} \cup \langle (l_2, 0 \leq x \wedge y < 0) \rangle^{\leftarrow}\end{aligned}$$

To compute $\langle (l_1, y < 0) \rangle^{\leftarrow}$ assume a time step resulting in a state from $(l_1, y < 0)$. 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 $\langle (l_1, y < 0) \rangle^{\leftarrow} = (l_1, y < 0)$.

To compute $\langle (l_2, 0 \leq x \wedge y < 0) \rangle^{\leftarrow}$ assume a time step resulting in a state from $(l_2, 0 \leq x \wedge y < 0)$. 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 $\langle (l_2, 0 \leq x \wedge y < 0) \rangle^{\leftarrow} = (l_2, 0 \leq x \wedge y < x)$.

Collecting the above information, $\langle R_{bad} \rangle^{\leftarrow} = (l_1, y < 0) \cup (l_2, 0 \leq x \wedge y < x)$.

Now we compute the discrete step predecessor of this set.

$$\begin{aligned}pre[\langle R_{bad} \rangle^{\leftarrow}] &= \\ pre[(l_1, y < 0) \cup (l_2, 0 \leq x \wedge y < x)] &= \\ \underbrace{(l_1, y < 0)}_{\tau_{l_1}} \cup \underbrace{(l_2, 0 \leq x \wedge y < x)}_{\tau_{l_2}} \cup \underbrace{(l_2, 0 \leq x \wedge y < 0)}_{\text{edge from } l_2 \text{ to } l_1} \cup \underbrace{(l_1, 0 \leq x \wedge y < x)}_{\text{edge from } l_1 \text{ to } l_2} &= \\ (l_1, y < 0 \vee 0 \leq y < x) \cup (l_2, (0 \leq x \wedge y < 0) \vee (0 \leq y < x)) &= \end{aligned}$$

The intersection of this predecessor set with R_1 yields

$$\begin{aligned} \text{pre}[\langle R_{bad} \rangle^{\prec}] \cap R_1 &= [(l_1, y < 0 \vee 0 \leq y < x) \cup (l_2, (0 \leq x \wedge y < 0) \vee (0 \leq y < x))] \cap \\ &\quad [(l_1, 0 \leq y) \cup (l_2, 0 \leq x \wedge 0 \leq y)] \\ &= (l_1, 0 \leq y < x) \cup (l_2, 0 \leq y < x) \\ &=: R_2. \end{aligned}$$

We define

$$R_3 := R_1 \setminus R_2 = (l_1, 0 \leq x \leq y) \cup (l_2, 0 \leq x \wedge 0 \leq y \wedge x \leq y).$$

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

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

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

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

In the next iteration we detect that the termination condition $\text{reach} = \text{completed}$ holds. Since $R_{bad} \notin \text{reach}$, the algorithm returns that the property holds.

The minimization of linear hybrid automata 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

General Hybrid Automata

In the previous chapter we have seen an approach for the reachability analysis of hybrid systems with linear behavior based on fixed point computation. There we represented the reachable sets by conjunctions of linear constraints. In this chapter we discuss the analysis of general hybrid automata, and especially other representation techniques for their state sets.

The reachability problem for hybrid automata is in general undecidable. Nevertheless, incomplete algorithms exist for reachability analysis, which allow to check safety properties of the systems. Most algorithms compute *over-approximations* of the reachability relation. Generally there are two kinds of approaches:

1. We can build a finite abstraction of the state space, and compute reachability for the abstract system. We have already seen a typical example for this approach: the region automata construction for timed automata. As this abstraction is based on bisimulation, it is not over-approximating. Another example is the on-the-fly refinement of the predicate abstraction during the fixed-point computation of the last chapter. This abstraction is in general over-approximating. Spurious counterexamples can be eliminated by (counterexample-guided) abstraction refinement (realized by splitting the abstract states).
2. An alternative approach is to compute reachability for the original system, without abstraction, but over-approximating the set of reachable states.

In this chapter we deal with the latter approach. The general forward reachability computation procedure can be specified by the algorithm depicted on Figure 7.1. Note that if the state space is infinite, then this algorithm does not always terminate.

In order to implement the above algorithm, we must solve two problems:

1. We must be able to store the current reachable sets R and R^{new} , and build their union, intersection, etc.
2. We must be able to compute $\text{Reach}(P)$ for a set P .

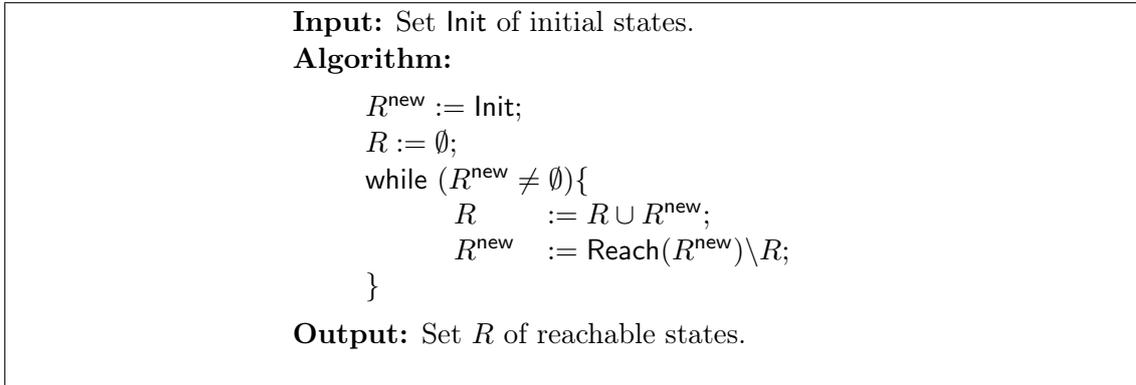


Figure 7.1: General reachability computation algorithm

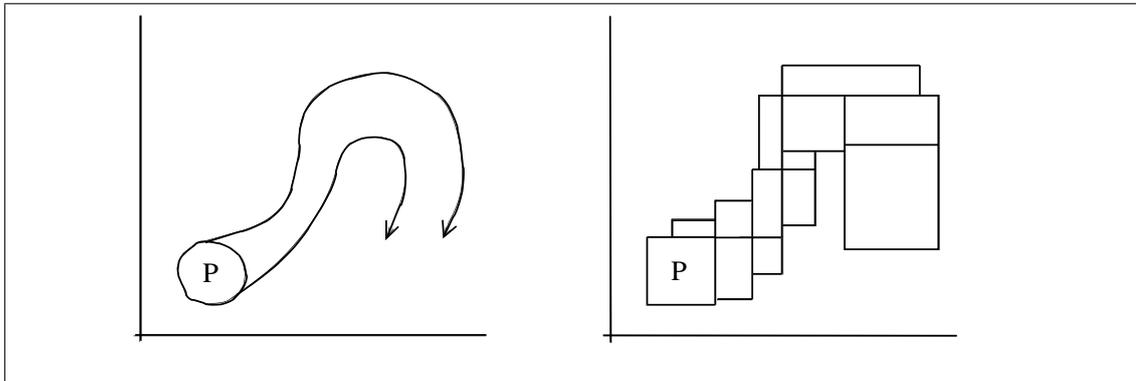


Figure 7.2: An example reachability approximation for a hybrid automaton

Note that in general the reachable sets cannot be represented exactly, as for general hybrid automata even the reachability relation due to a single time step can be undecidable. We solve the above problems by

1. over-approximating the reachable sets by sets having certain geometric forms, such that the needed operations can be done efficiently, and
2. over-approximate $\text{Reach}(P)$ in each step in the above procedure.

In this chapter we discuss the first point, and introduce representations of state sets. Afterwards, in the next chapter we discuss methods for the over-approximation of $\text{Reach}(P)$. Putting the two together, we can implement the algorithm on Figure 7.1. To get an intuition, an example reachability approximation using the reachability algorithm for some hybrid automaton is visualized on Figure 7.2. The exact continuous behavior is depicted on the left, while the approximation is drawn on the right.

The *geometry* chosen to represent reachable sets has a crucial effect on the practicability of the whole procedure. Usually, the more complex the geometry,

1. the more costly is the *storage* of the sets,

2. the more difficult it is to *perform operations* like union and intersection, and
3. the more elaborate is the *computation of new reachable* sets, but
4. the better the *approximation* of the set of reachable states.

Choosing the geometry has to be a *compromise* between these impacts. The representation should allow *efficient computation* of the operations for

- membership relation,
- union,
- intersection,
- subtraction,
- test for emptiness.

In the remaining part of this chapter we have a closer look at representation by

- orthogonal polyhedra in Section 7.1 and
- convex polyhedra in Section 7.2.

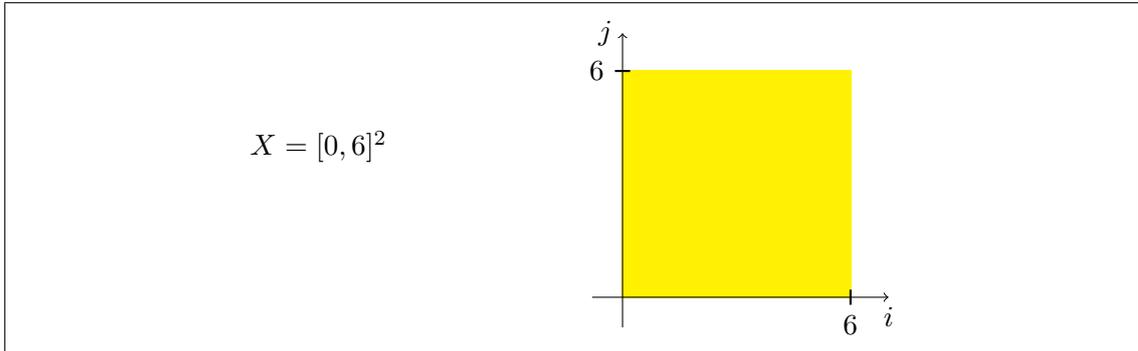


Figure 7.3: An example domain

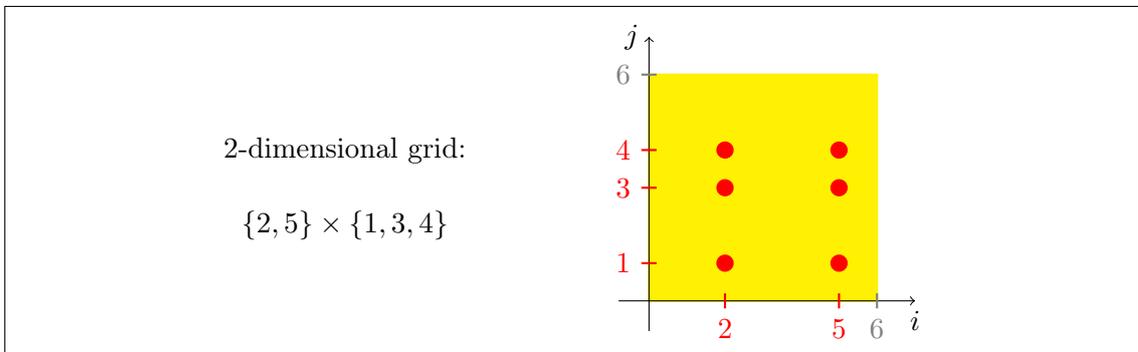


Figure 7.4: An example grid

7.1 Orthogonal Polyhedra

The content of this section is based on the publications [BMP99] and [SK03].

7.1.1 Definition of orthogonal polyhedra

X As state space *domain* we consider a bounded subset $X = [0, m]^d \subseteq \mathbb{R}^d$ ($m \in \mathbb{N}_+$) of the reals (can be extended to $X = \mathbb{R}_+^d$). Elements of X are denoted by $x = (x_1, \dots, x_d)$, the zero vector by $\mathbf{0}$, and the unit vector by $\mathbf{1}$. For our running example we define the domain $X = [0, 6]^2$, depicted on Figure 7.3.

Definition 45.

A d -dimensional grid associated with the domain $X = [0, m]^d \subseteq \mathbb{R}^d$ ($m \in \mathbb{N}_+$) is a product of d subsets of $\{0, 1, \dots, m-1\}$.

An example grid is depicted on Figure 7.4.

Definition 46.

\mathbf{G} The elementary grid associated with $X = [0, m]^d \subseteq \mathbb{R}^d$ ($m \in \mathbb{N}_+$) is $\mathbf{G} = \{0, 1, \dots, m-1\}^d \subseteq \mathbb{N}^d$.

An example elementary grid is depicted on Figure 7.5 The grid admits a natural *partial order* with $(m-1, \dots, m-1)$ on the top and $\mathbf{0}$ as bottom, as show on Figure 7.6.

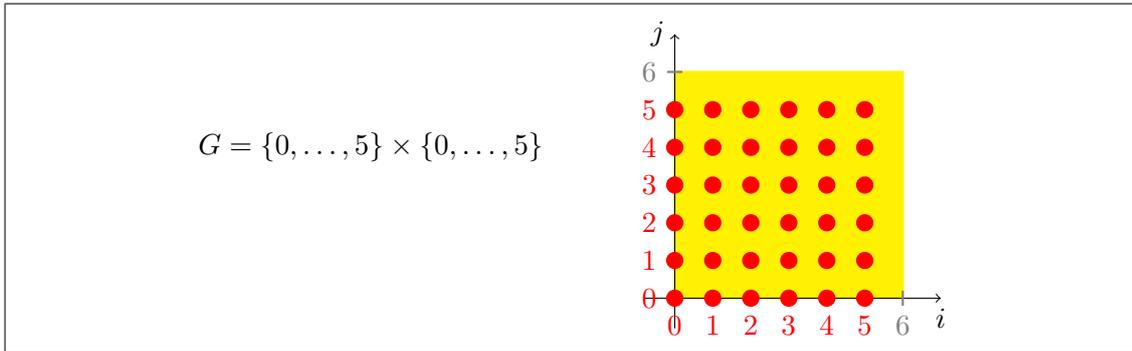


Figure 7.5: An example elementary grid

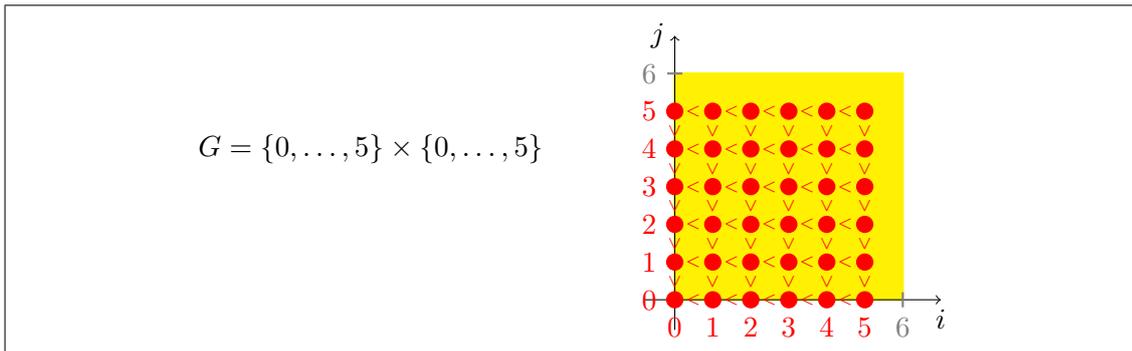


Figure 7.6: The partial order on the grid points of an elementary grid

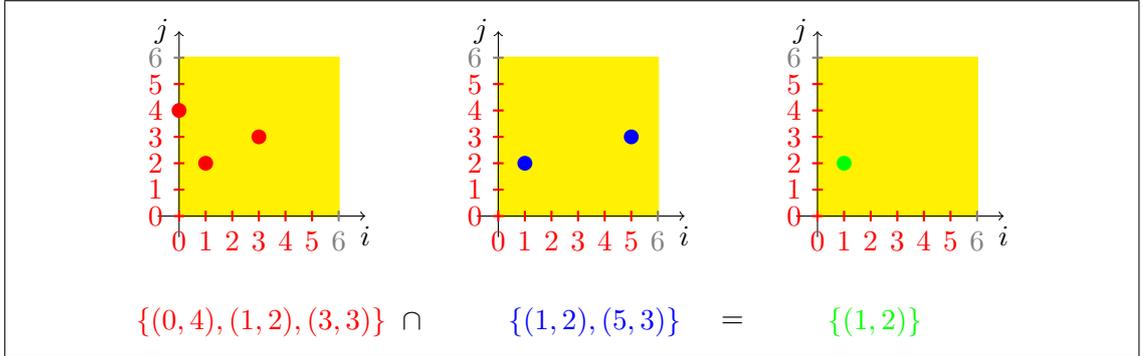


Figure 7.7: An example cut of two subsets of an elementary grid

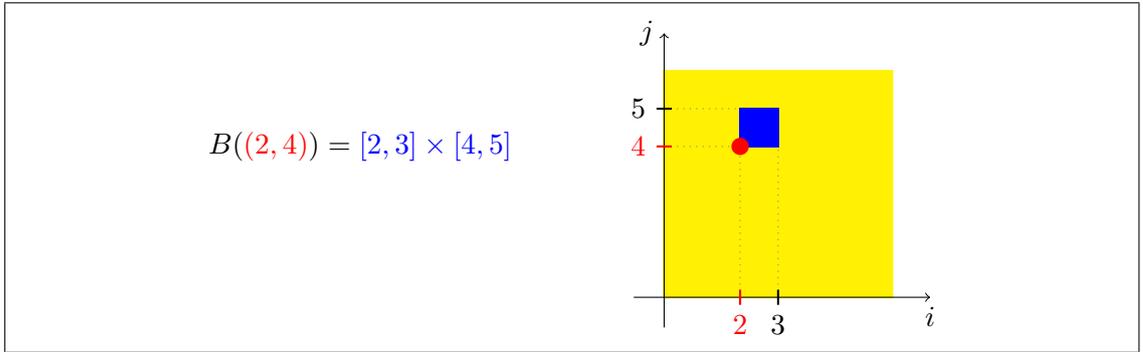


Figure 7.8: The elementary box of a grid point

The set of subsets of the elementary grid \mathbf{G} forms a *Boolean algebra* $(2^{\mathbf{G}}, \cap, \cup, \sim)$ under the set-theoretic operations

- $A \cup B$
- $A \cap B$
- $\sim A = \mathbf{G} \setminus A$

for $A, B \subseteq \mathbf{G} \subset \mathbb{N}^d$. An example cut of two subsets is shown on Figure 7.7.

Definition 47 (Elementary box).

B, \mathbf{B} The elementary box associated with a grid point $x = (x_1, \dots, x_d)$ is $B(x) = [x_1, x_1 + 1] \times \dots \times [x_d, x_d + 1]$. The set of elementary boxes is denoted by \mathbf{B} .

An example of an elementary box of a grid point is shown on Figure 7.8.

Definition 48 (Orthogonal polyhedra).

An orthogonal polyhedron P is a union of elementary boxes, i.e., an element of $2^{\mathbf{B}}$.

An example of an orthogonal polyhedron is shown on Figure 7.9.

The set of orthogonal polyhedra forms a *Boolean algebra* $(2^{\mathbf{B}}, \sqcap, \sqcup, \neg)$ with the operations

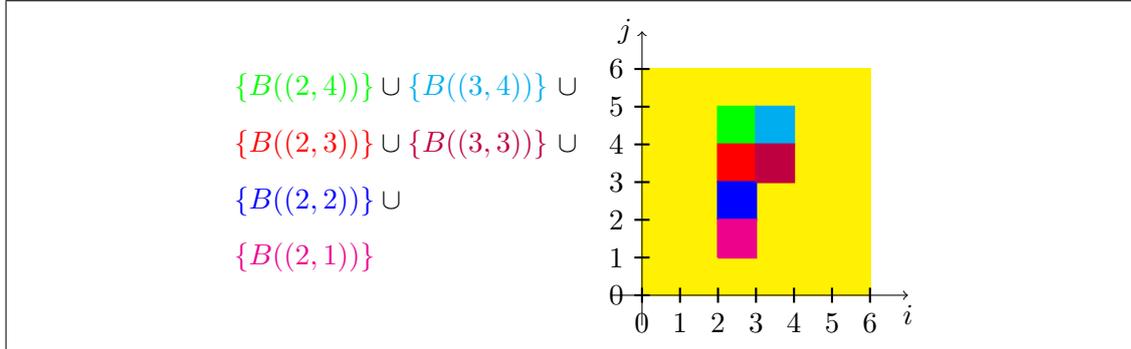


Figure 7.9: An example orthogonal polyhedron

- $A \sqcup B = A \cup B$
- $A \sqcap B = cl(int(A) \cap int(B))$
- $\neg A = cl(X \setminus A)$

where

- int is the interior operator yielding the largest open set $int(A)$ contained in A , and
- cl is the topological closure operator yielding the smallest closed set $cl(A)$ containing A .

For the operations \sqcap and $\neg A$ we need the interior and closure operations in order to be closed under orthogonal polyhedra, i.e., for the result being an orthogonal polyhedra. Intuitively, we use the standard set operations but want to compute the cut and negation not in terms of points but in terms of elementary boxes.

Figure 7.10 gives an example for \sqcap . Without the interior operation the result would be the line between the two elementary boxes. With the interior operation the result is the empty set.

Figure 7.11 gives an example for the negation. Without building the closure of the set negation the result would be an open set.

The *bijection between \mathbf{G} and \mathbf{B}* which associates every elementary box with its leftmost corner generates an *isomorphism between $(2^{\mathbf{G}}, \cap, \cup, \sim)$ and $(2^{\mathbf{B}}, \sqcap, \sqcup, \neg)$* . Thus we can switch between point-based and box-based terminology according to what serves better the intuition, as illustrated on Figure 7.12.

7.1.2 Representation of orthogonal polyhedra

We need an representation of orthogonal polyhedra that allows efficient computation. The representations we consider are based on the vertices of orthogonal polyhedra. Below we give some definitions we need for the definition of a vertex.

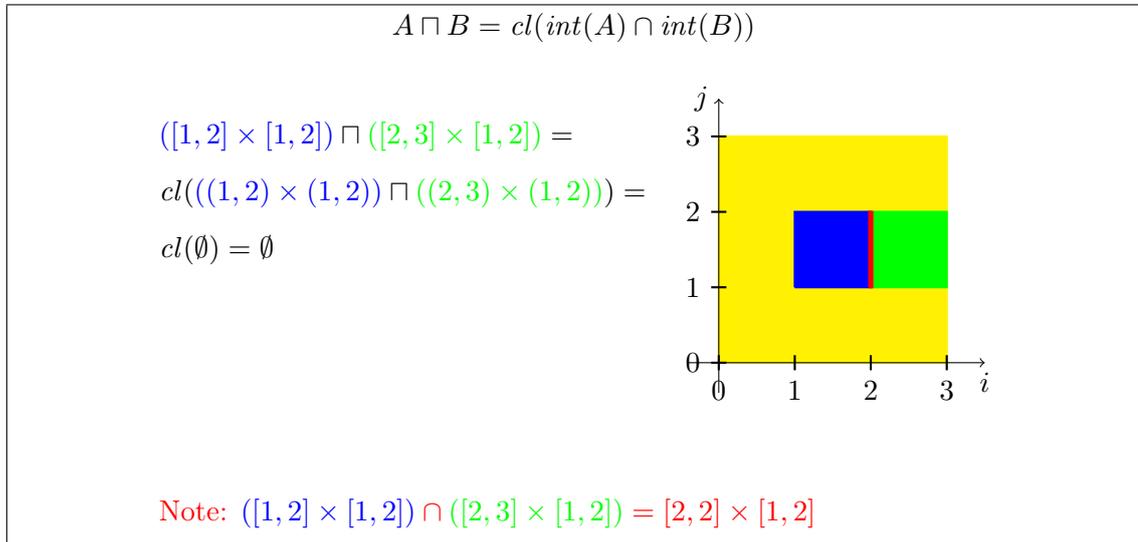


Figure 7.10: An example for the cut of two orthogonal polyhedra

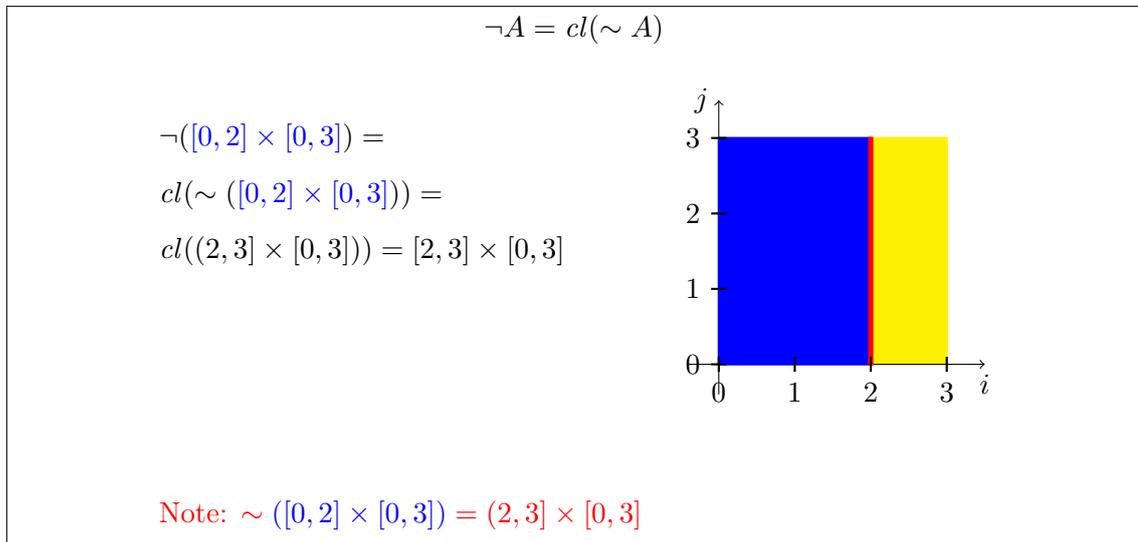


Figure 7.11: An example for the negation of an orthogonal polyhedron

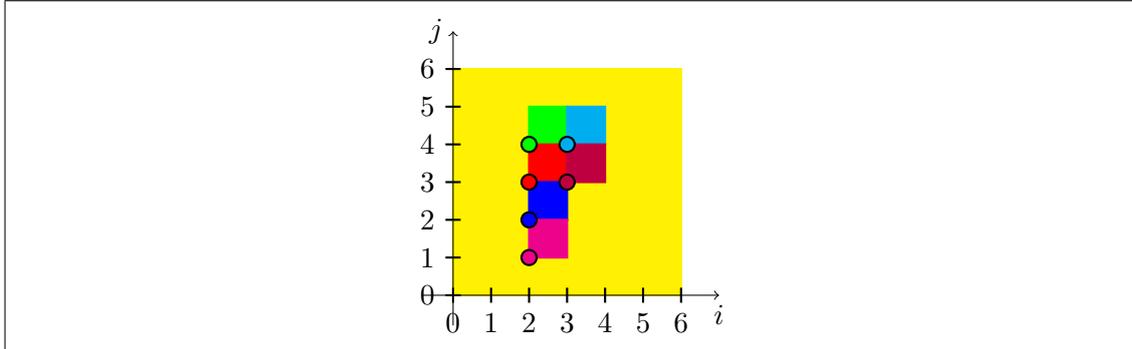


Figure 7.12: Bijection between \mathbf{G} and \mathbf{B}

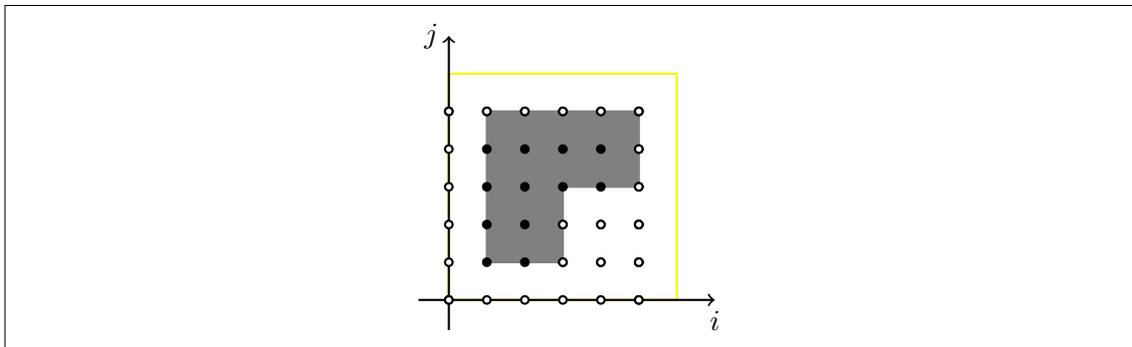


Figure 7.13: The coloring function for an orthogonal polyhedron

Definition 49 (Color function).

Let P be an orthogonal polyhedron. The color function $c : X \rightarrow \{0, 1\}$ is defined by

$$c(x) = \begin{cases} 1 & \text{if } x \text{ is a grid point and } B(x) \subseteq P \\ 0 & \text{otherwise} \end{cases}$$

for all $x \in X$.

If $c(x) = 1$ we say that x is black and that $B(x)$ is full.

If $c(x) = 0$ we say that x is white and that $B(x)$ is empty.

Note that c almost coincides with the characteristic function of P as a subset of X . It differs from it only on right-boundary points. The coloring for an example orthogonal polyhedron is shown in Figure 7.13.

The following definitions capture the intuitive meaning of a facet and a vertex and, in particular, that the boundary of an orthogonal polyhedron is the union of its facets.

Definition 50 (i -predecessor).

The i -predecessor of a grid point $x = (x_1, \dots, x_d) \in X$ is $x^{i-} = (x_1, \dots, x_{i-1}, x_i - 1, x_{i+1}, \dots, x_d)$. We use x^{ij-} to denote $(x^{i-})^{j-}$. When x has no i -predecessor, we write \perp for the predecessor value.

The above definition is illustrated in Figure 7.14.

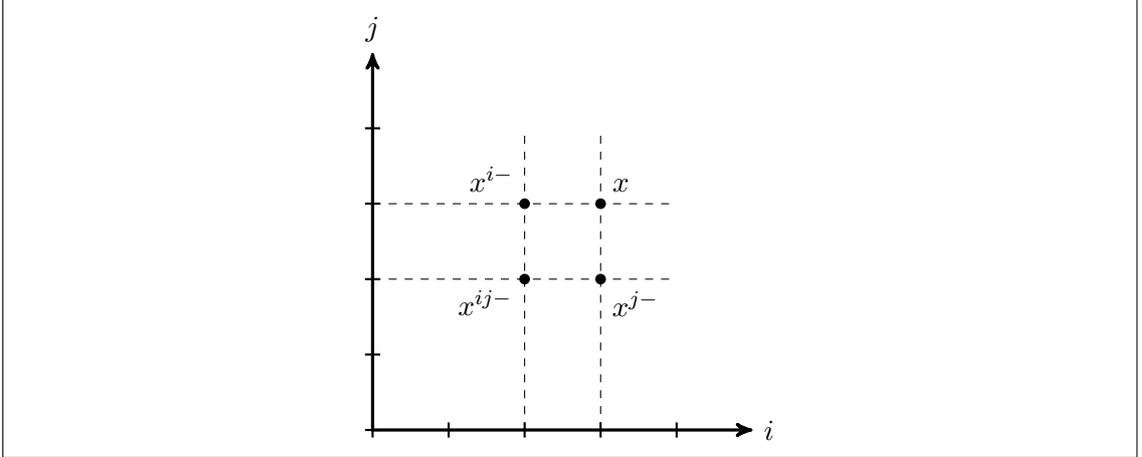


Figure 7.14: The predecessors of a grid point

Definition 51 (Neighborhood).

The neighborhood of a grid point x is the set

$$\mathcal{N}(x) = \{x_1 - 1, x_1\} \times \dots \times \{x_d - 1, x_d\}$$

(the vertices of a box lying between $x - \mathbf{1}$ and x). For every i , $\mathcal{N}(x)$ can be partitioned into left and right i -neighborhoods

$$\mathcal{N}^{i-}(x) = \{x_1 - 1, x_1\} \times \dots \times \{x_i - 1\} \times \{x_d - 1, x_d\}$$

and

$$\mathcal{N}^i(x) = \{x_1 - 1, x_1\} \times \dots \times \{x_i\} \times \{x_d - 1, x_d\}.$$

Definition 52 (i -hyperplane).

An i -hyperplane is a $(d - 1)$ -dimensional subset $H_{i,z}$ of X consisting of all points x satisfying $x_i = z$.

The above definition is illustrated in Figure 7.15.

Note that the facets of orthogonal polyhedra are $d - 1$ -dimensional polyhedra, and as such, they are subsets of i -hyperplanes. For the definition of facets we make use of the fact that the coloring changes on facets. Thereby we need to pay attention to white vertices lying on the boundaries to the “right”. For example, the orthogonal polyhedron in Figure 7.13 has white grid points on the top and at the right that belong to facets or are vertices.

Definition 53 (i -facet).

An i -facet of an orthogonal polyhedron P with color function c is

$$F_{i,z}(P) = cl\{x \in H_{i,z} \mid c(x) \neq c(x^{i-})\}$$

for some integer $z \in [0, m)$.

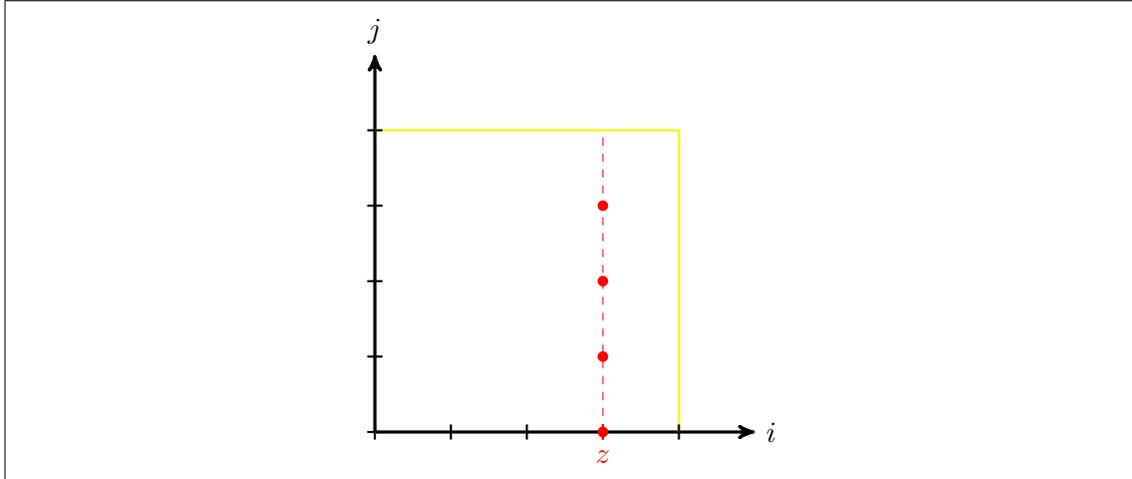


Figure 7.15: An i -hyperplane of X

Definition 54 (Vertex).

A vertex is a non-empty intersection of d distinct facets. The set of vertices of an orthogonal polyhedron P is denoted by $V(P)$.

In Figure 7.16 the vertices of an orthogonal polyhedron are marked red.

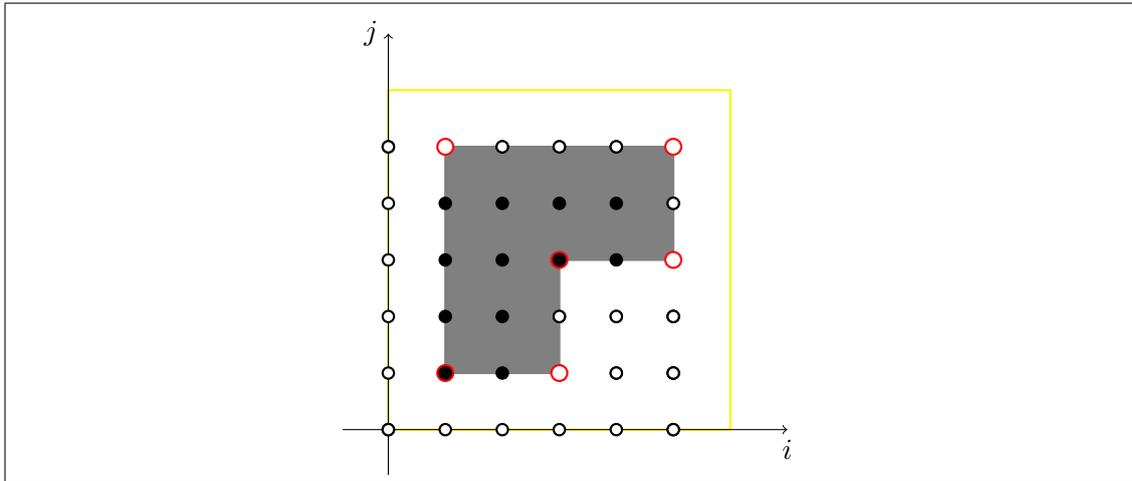


Figure 7.16: The vertices of an orthogonal polyhedron

Definition 55 (i -vertex-predecessor).

An i -vertex-predecessor of a grid point $x = (x_1, \dots, x_d) \in X$ is a vertex of the form $(x_1, \dots, x_{i-1}, z, x_{i+1}, \dots, x_d)$ for some integer $z \in [0, x_i]$. When x has no i -vertex-predecessor, we write \perp for its value.

The first i -vertex-predecessor of x , denoted by $x^{i\leftarrow}$, is the one with the maximal z .

Figure 7.17 shows the first i -vertex-predecessors of some points for our example orthogonal polyhedron.

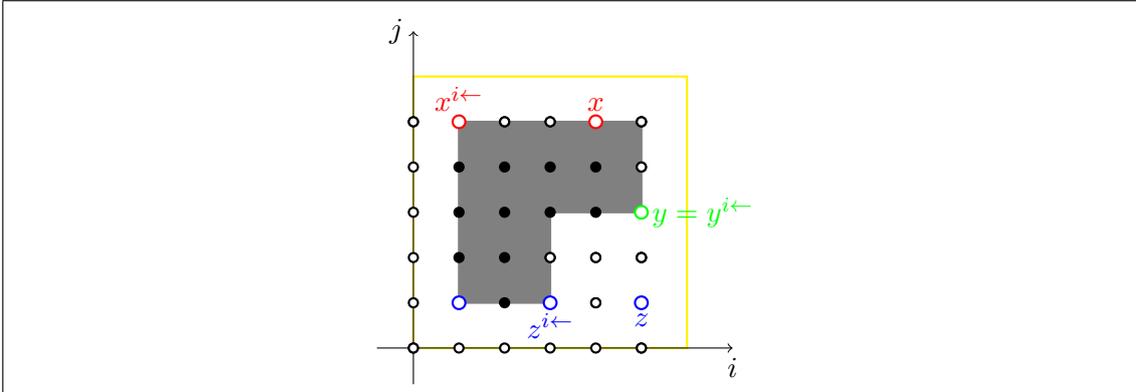


Figure 7.17: The first i -vertex-predecessors of some points and an orthogonal polyhedron

A *representation scheme* for $2^{\mathbf{B}}$ ($2^{\mathbf{G}}$) is a set \mathcal{E} of syntactic objects such that there is a surjective function ϕ from \mathcal{E} to $2^{\mathbf{B}}$, i.e., every syntactic object represents at most one polyhedron and every polyhedron has at least one corresponding object. If ϕ is an injection we say that the representation is *canonical*, i.e., every polyhedron has a unique representation.

There are different representation schemes we could consider. A naive way would be an *explicit representation* consisting of the enumeration of the color values on every grid point, i.e., a d -dimensional zero-one array with m^d entities. However, such a representation would be both for storage and computation very inefficient.

Another possibility would be a *Boolean representation*, specifying an orthogonal polyhedron as a Boolean combination of inequalities of the form $x_i \geq z$. Note that this representation is non-canonical.

In this section we consider the following vertex-based representations:

- *Vertex representation:* An orthogonal polyhedron P is represented by the set $\{(x, c(x)) \mid x \text{ is a vertex of } P\}$, i.e., the vertices of P along with their color. This representation is canonical.

Note that the vertices alone would not yield a unique representation scheme. Figure 7.18 shows two orthogonal polyhedra having the same vertex sets but assigning different colors to the vertices.

Note also that not every set of points and colors is a valid representation of a polyhedron.

- *Neighborhood representation:* An orthogonal polyhedron is represented by its vertices along with the colors of all the 2^d points in the neighborhoods of the vertices.
- *Extreme vertex representation:* This representation keeps the vertices and the *parity* of the number of black points in the neighborhood of the vertices. In fact, it suffices to keep only vertices with odd parity.

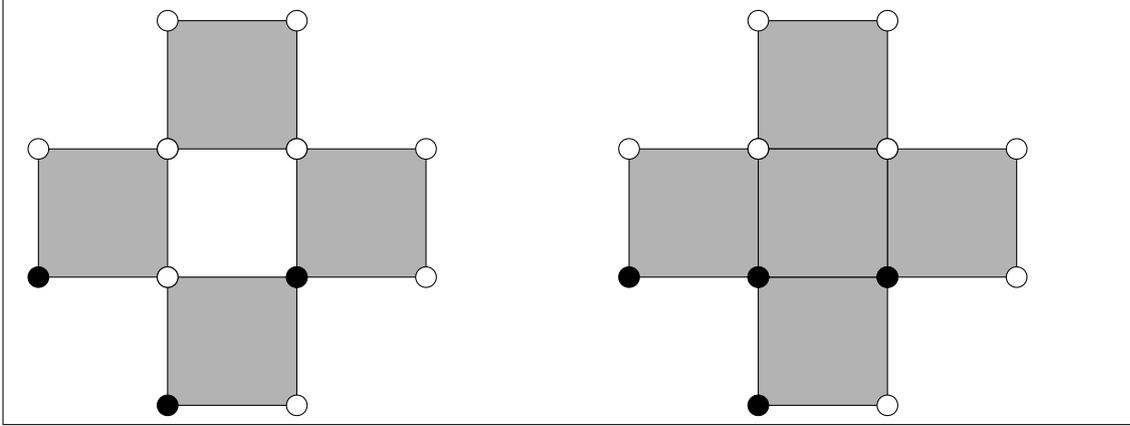


Figure 7.18: Two vertex representations agreeing on the vertices but having different vertex colors

Why we choose the above representations, especially in the case of the neighborhood and the extreme vertex representations, will become clear in the following when we define the algorithms for the required operations on orthogonal polyhedra. We do not discuss all needed operations. Instead, we describe the algorithms for the membership problem and for the computation of the intersection of two orthogonal polyhedra, only.

7.1.3 Membership problem

Next we discuss how to solve the *membership problem* for orthogonal polyhedra, based on the three different representation schemes suggested above. Given a representation of a polyhedron P and a grid point x , the membership problem is the problem to determine $c(x)$, that is, whether $B(x) \subseteq P$.

Membership problem for the vertex representation

We make use of the following observations:

- A point x is on an i -facet iff

$$\exists x' \in \mathcal{N}^i(x). c(x'^{-}) \neq c(x').$$

- A point x is a vertex iff

$$\forall i \in \{1, \dots, d\}. \exists x' \in \mathcal{N}^i(x). c(x'^{-}) \neq c(x').$$

- A point x is not a vertex iff

$$\exists i \in \{1, \dots, d\}. \forall x' \in \mathcal{N}^i(x). c(x'^{-}) = c(x').$$

Given an orthogonal polyhedron in vertex representation, we can compute the color of a non-vertex grid point recursively, using the following lemma:

Lemma 14 (Color of a non-vertex).

Let x be a non-vertex. Then there exists a direction $j \in \{1, \dots, d\}$ such that

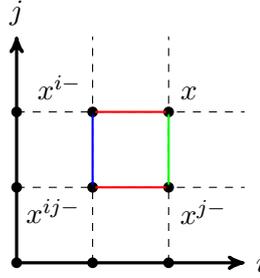
$$(7.1) \quad \forall x' \in \mathcal{N}^j(x) \setminus \{x\}. c(x'^{j-}) = c(x').$$

Let j be such a direction. Then $c(x) = c(x^{j-})$.

Proof A point x is not a vertex iff there is a dimension in that the point is not on a facet, i.e.,

$$(7.2) \quad \exists i \in \{1, \dots, d\}. \forall x' \in \mathcal{N}^i(x). c(x'^{i-}) = c(x').$$

Thus j always exists. Assume a j satisfying 7.1 and an i satisfying 7.2. If $i = j$, then the case is straightforward. Otherwise, if $i \neq j$, for i we have $c(x^{i-}) = c(x)$ and $c(x^{ij-}) = c(x^{j-})$. For j we have $c(x^{ij-}) = c(x^{i-})$. By the transitivity of “=” we get $c(x) = c(x^{j-})$.



Consequently we can calculate the color of a non-vertex x based on the color of all points in $\mathcal{N}(x) - \{x\}$: just find some j satisfying the conditions of the above lemma and let $c(x) = c(x^{j-})$.

Theorem 3.

For a domain $X = [0, n]^d$, the membership problem for vertex representation can be solved in time $\mathcal{O}(n^d d 2^d)$ using space $\mathcal{O}(n^d)$.

Proof We start at x and recursively determine the membership of all the $2^d - 1$ points in $\mathcal{N}(x) \setminus \{x\}$. Termination of the recursion is guaranteed because we go down in the partial order on $2^{\mathbb{G}}$ and either encounter vertices or reach the origin. We must recursively determine the color of at most n^d grid points. For each of them we must check at most d dimensions if they satisfy the condition of the lemma on the color of a non-vertex. Checking the condition in a dimension invokes $2^{d-1} - 1$ color comparisons. ■

However, this algorithm is not very efficient, because in the worst-case one has to calculate the color of all the grid points between $\mathbf{0}$ and x . We can improve it using the notion of an *induced grid*: let the i -scale of P be the set of the i -coordinates of the vertices of P , and let the induced grid be the Cartesian product of its i -scales. The induced grid is the smallest (coarsest) grid containing all the vertices. Every rectangle in the

induced grid has a uniform color. Calculating the color of a point reduces to finding its closest “dominating” point on the induced grid and applying the algorithm to that grid in $\mathcal{O}(n^d d 2^d)$ time, where n is the number of grid points in the induced grid. The approach is illustrated in Figure 7.19.

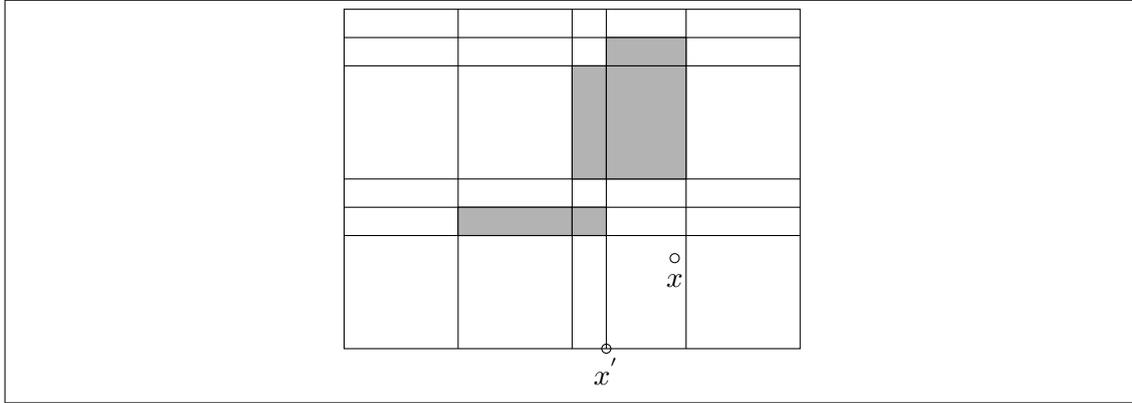


Figure 7.19: Membership problem on an induced grid

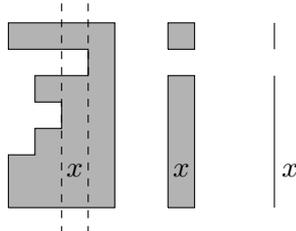
Membership problem for the neighborhood representation

The solution of the membership problem for the neighborhood representation is based on projection.

Definition 56 (*i*-slice and *i*-section).

Let P be an orthogonal polyhedron and z an integer in $[0, m)$.

- The *i*-slice of P at z is the d -dimensional orthogonal polyhedron $J_{i,z}(P) = P \cap \{x \mid z \leq x_i \leq z + 1\}$.
- The *i*-section of P at z is the $(d - 1)$ -dimensional orthogonal polyhedron $\mathcal{J}_{i,z}(P) = J_{i,z}(P) \cap H_{i,z}$.



The membership of $x = (x_1, \dots, x_d)$ can be reduced to membership in $\mathcal{J}_{i,x_i}(P)$, which is a $(d - 1)$ -dimensional problem: a x is contained in an orthogonal polyhedron P iff it is contained in the *i*-section of P at x_i . By successively reducing dimensionality for every i we obtain a point whose color is that of x .

For the computation of an *i*-section of an orthogonal polyhedron P we first observe that the vertices of the *i*-section are points x in the corresponding *i*-hyperplane that lie on a

facet of P in each dimension $j \neq i$. I.e., a point x in the corresponding i -hyperplane is a vertex of the i -section iff (1) x has a first i -vertex-predecessor $y = x^{i\leftarrow} \neq \perp$ such that (2) for each dimension $j \neq i$ there is a j -facet to the right of y in dimension i . The second condition assures that this facet intersects with the i -hyperplane.

The neighborhood representation provides us with the vertices and the colors of their neighbors. This coloring information can be used to determine in which directions (relative to the vertex) the faces defining the vertex lie and thus to check the second condition. Let us take as an example the two-dimensional case. Figure 7.20 lists all 16 possible neighborhood colorings of a grid point x for $d = 2$. The colorings in the first row define non-vertex grid points, whereas the second and third rows contain vertices. Let i be the horizontal and j the vertical dimension. For the vertices, i.e., the points in the second and third rows, we can observe that there is a j -facet to the right of x (i.e., to the right in dimension i) iff the color of x differs from the color of x^{j-} , i.e., the grid point below x (the point to the left from x in dimension j). This is the case for all grid points in the third row, whereas the grid points of the second row do not fulfill this condition.

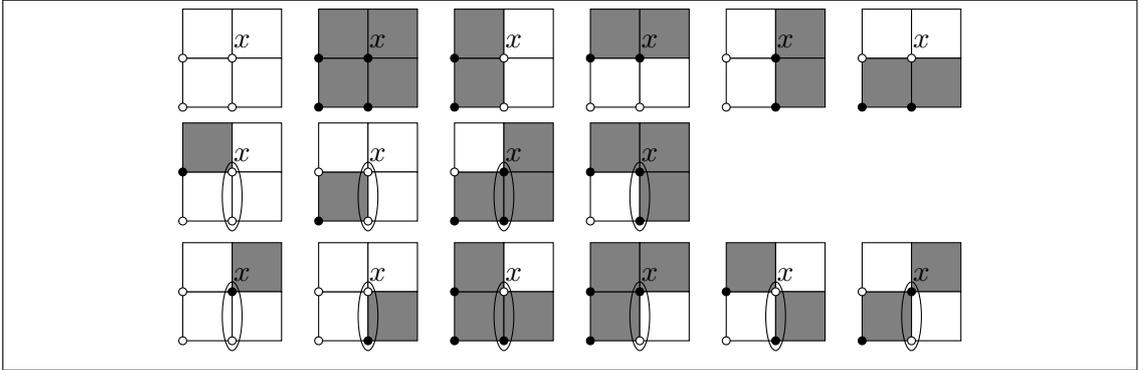


Figure 7.20: Possible neighborhood colorings in the two-dimensional case

We can use this information to define a projection operation. We introduce an $\mathcal{O}(n \log n)$ membership algorithm for the neighborhood representation, based on successive projections of P into polyhedra of smaller dimension.

We use the following lemma to calculate the neighborhood representation of an i -section.

Lemma 15 (Vertex of a section).

Let P be an orthogonal polyhedron and let P' be its i -section at $x_i = z$. A point x is a vertex of P' iff $y = x^{i\leftarrow} \neq \perp$ and for every $j \neq i$ there exists $x' \in \mathcal{N}^i(y) \cap \mathcal{N}^j(y)$ such that $c(x'^{j-}) \neq c(x')$.

Moreover, when this condition is true, the neighborhood of x relative to $\mathcal{J}_{i,z}(P)$ is given by $\mathcal{N}^i(y)$.

Proof Assume x is a vertex of P' . Then there is a facet of P orthogonal to i that goes through x . This is possible only if there is a vertex $x^{i\leftarrow} = y = (x_1, \dots, x_{i-1}, z, x_{i+1}, \dots, x_d)$ left to x in the dimension i . The vertex y lies on a facet in all dimensions $j \neq i$, implying the existence of an $x' \in \mathcal{N}^i(y) \cap \mathcal{N}^j(y)$ such that $c(x'^{j-}) \neq c(x')$. Finally, since the facet goes from y to the right in i through x , we have that $c(\mathcal{N}^i(y)) = c(\mathcal{N}^i(x))$.

Assume conversely $y = x^{i\leftarrow}$ exists and it satisfies the condition. Then $c(\mathcal{N}^i(x)) = c(\mathcal{N}^i(y))$, because otherwise, by the above reasoning, there would be a vertex between x and y . Hence x satisfies the condition. ■

The resulting projection algorithm is illustrated on an example in Figure 7.21.

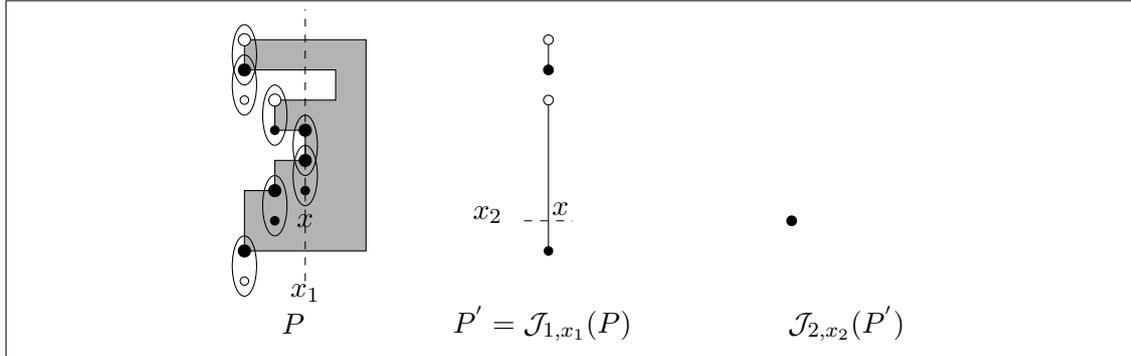


Figure 7.21: Solving the membership problem by projection for the neighborhood representation

Theorem 4 (Membership problem for the neighborhood representation).

The membership problem for the neighborhood representation can be solved in time $\mathcal{O}(nd^2(\log n + 2^d))$.

Proof For a d -dimensional orthogonal polyhedron P with n vertices we can determine those vertices y which are $x^{i\leftarrow}$ for some $x \in H_{i,z}$ in $\mathcal{O}(nd \log n)$ steps.

There are most n such points. With the above lemma we can determine in $\mathcal{O}(d2^d)$ time whether one such point is a vertex of the section. Hence it takes $\mathcal{O}(nd(\log n + 2^d))$ to get rid of one dimension.

This is repeated d times until P is contracted into a point. ■

A similar algorithm with the same complexity can be used to calculate the *color of all the points in a neighborhood of x* . The algorithm takes double slices (d -dimensional thick sections of width two) of P , and successively reduces P into the neighborhood of x . This variation of the algorithm is used for doing Boolean operations.

Membership problem for the extreme vertex representation

The *extreme vertex representation* can be viewed as a compaction of the neighborhood representation. Instead of maintaining all the neighborhood of each vertex, it suffices to keep only the *parity* of the number of black points in that neighborhood – in fact it suffices to keep only vertices with odd parity.

Definition 57 (Extreme point).

We use $\text{parity}(x)$ to denote the parity of the number of black points in the neighborhood $\mathcal{N}(x)$ of a grid point x . The grid point x is said to be extreme if $\text{parity}(x) = 1$.

Lemma 16.

An extreme point is a vertex.

Proof By induction on the dimension d . The base case $d = 1$ is immediate. For $d > 1$, choose $i \in \{1, \dots, d\}$. Exactly one of $\mathcal{N}^{i-}(x)$ and $\mathcal{N}^i(x)$ contains an odd number of black points. Assume w.l.o.g. that it is $\mathcal{N}^i(x)$. By induction hypothesis x is a vertex in $\mathcal{J}_{i,x_i}(P)$. I.e., for every $j \neq i$ there exists $x' \in \mathcal{N}^j(x)$ such that $c(x'^{j-}) \neq c(x')$. Since one cannot have $c(x') = c(x'^{i-})$ for all $x' \in \mathcal{N}^i(x)$, x is a vertex of P . ■

The converse is not true, i.e., vertices in general need not be extreme.

An *extreme vertex representation* consists in representing an orthogonal polyhedron by the set of its extreme vertices. (Additionally, the color of the origin is stored in a bit. From this information the colors of all extreme vertices can be inferred.)

Note that for $d = 1$ all vertices are extreme and hence the vertex and extreme vertex representations coincide. Figure 7.22 gives some two-dimensional representation examples, where the extreme vertices of the objects are marked red. For the objects in the left and in the middle all vertices are extreme. For the object in the right all 4 vertices in the middle are not extreme.

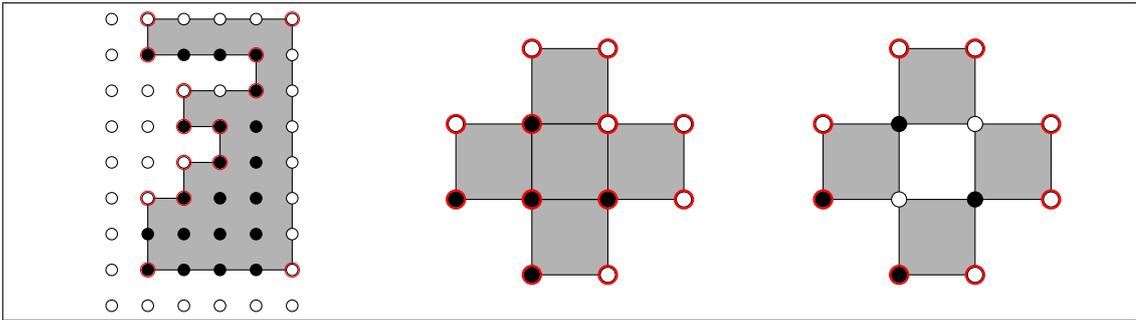


Figure 7.22: Examples for the extreme vertex representation

The *membership problem* is solved again by *projection*. To define a projection operation we need again a rule to determine which points of an i -section are extreme vertices of the projection.

Let us again first give an intuition in the two-dimensional case for the role of the parity of a vertex in the projection. The last two lines in Figure 7.20 list all possible neighborhoods of vertices of a two-dimensional orthogonal polyhedron. All but the last two vertices are extreme. Note that these last two cases are the only ones representing vertices where two facets meet each other.

Intuitively, the basic idea for the solution of the membership problem for the extreme vertex representation is the following: We again use projection. Given a point x on the i -hyperplane, we count its extreme i -vertex-predecessors to determine if x is an extreme vertex of the i -section. If there is an even number $2n$ of such vertices, then we can conclude that n facets started and ended at those vertices but there is no “open” facet to the left in i and thus the considered point x does not lie on any facet that goes orthogonal through the i -hyperplane. Consequently, x is not a vertex of the i -section. Otherwise,

if there is an odd number of such extreme i -vertex-predecessors, we can conclude that there is a facet going orthogonal through the i -hyperplane and x lies on that a facet. Furthermore, since the first i -extreme-vertex-predecessor of x lies on facets in all other dimensions $j \neq i$, it holds also for x , and thus x is a vertex of the i -section.

Lemma 17 (Extreme vertices of a section).

Let P be an orthogonal polyhedron and let $P' = \mathcal{J}_{i,z}(P)$. A point x is an extreme vertex of P' iff it has an odd number of extreme i -vertex-predecessors.

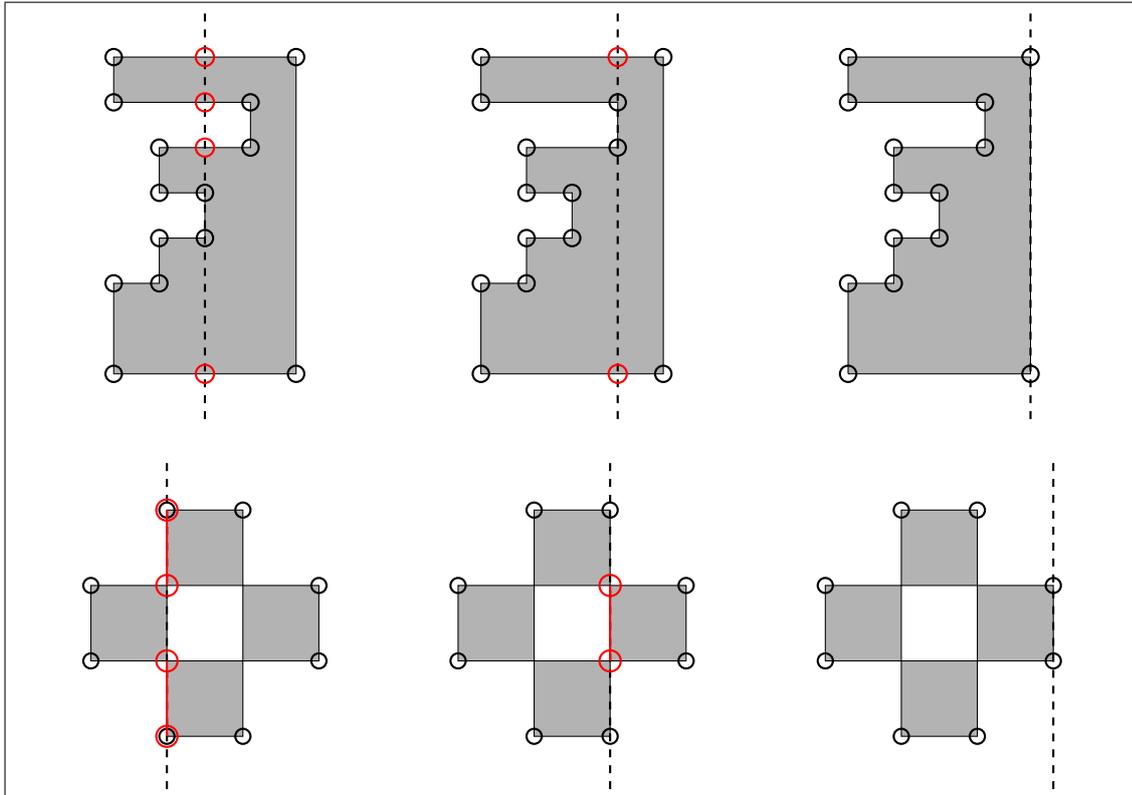


Figure 7.23: Extreme vertices of sections

7.1.4 Computing the Intersection of Orthogonal Polyhedra

We assume two polyhedra P_1 and P_2 with n_1 and n_2 vertices, respectively. The intersection of P_1 and P_2 may have vertices of P_1 and P_2 as well as some new vertices, as illustrated in Figure 7.24.

Lemma 18.

Let $x \in \mathbf{G}$ be a vertex of $P_1 \cap P_2$. Then for every dimension i , x is on an i -facet of P_1 or on an i -facet of P_2 .

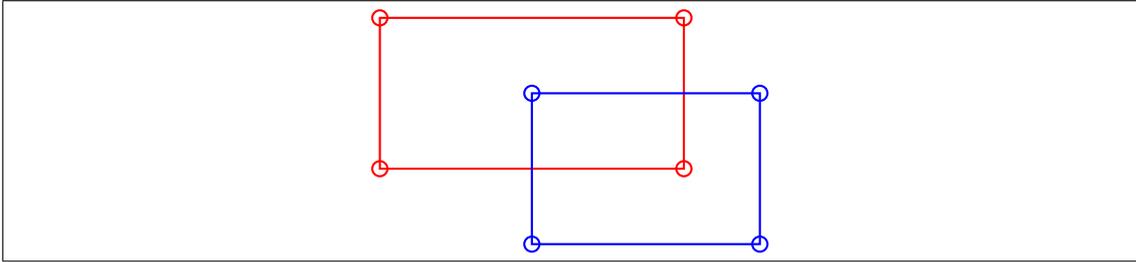


Figure 7.24: The intersection of two orthogonal polyhedra

That means, each “new” vertex of the intersection, not being a vertex of P_1 or P_2 , must lie in each dimension i on an i -facet of one of the intersecting polyhedra. Since in all three representations we have information about the vertices but not explicitly about the facets, the computation of the intersection of facets must be based on the available information about the vertices.

Lemma 19.

Let x be a vertex of $P_1 \cap P_2$ which is not an original vertex.

Then there exists a vertex y_1 of P_1 and a vertex y_2 of P_2 such that $x = \max(y_1, y_2)$, where \max is applied componentwise.

Note that due to symmetry, we could also use the minimum instead of the maximum in the above lemma.

Thus the candidates for being vertices of $P_1 \cap P_2$ are restricted to members from the set

$$V(P_1) \cup V(P_2) \cup \{x | \exists y_1 \in V(P_1). \exists y_2 \in V(P_2). x = \max(y_1, y_2)\},$$

whose number is not greater than $n_1 + n_2 + n_1 n_2$.

Figure 7.25 shows those candidates for the example intersection of Figure 7.24.

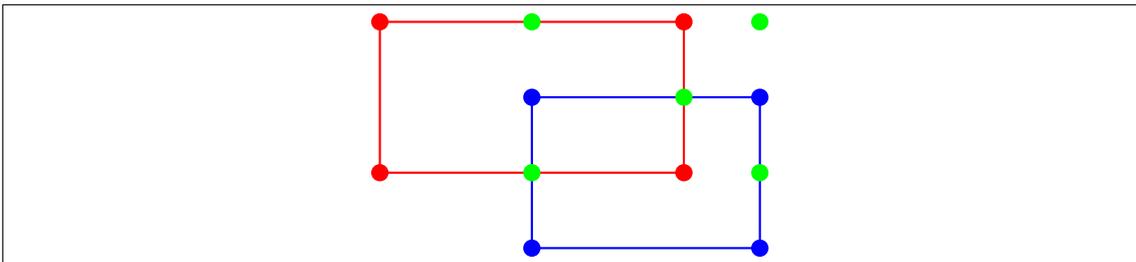


Figure 7.25: Vertex candidates of an intersection of two orthogonal polyhedra

The algorithm for computing the intersection of two polyhedra P_1 and P_2 works as follows:

- Initialize $V(P_1) \cup V(P_2)$ as the set of potential vertices of the intersection.
- For every pair of vertices calculate their max and add it to the potential vertex set.
- For each point in the potential vertex set:

- Compute the color of its neighborhood in both P_1 and P_2 .
- Calculate the intersection of the neighborhood coloring pointwise.
- Use the vertex rules to determine, whether the point is a vertex of the intersection.

Remember the vertex rule: A point x is a vertex iff

$$\forall i \in \{1, \dots, d\}. \exists x' \in \mathcal{N}^i(x). c(x'^{i-}) \neq c(x').$$

Figure 7.26 shows the computation for our example. The first picture shows the candidates. The next 4 rows illustrate the check of 4 of the candidates. The last row shows the result.

The algorithm for the intersection computation works similarly for all 3 representations. The only difference occurs in the computation of the colors for the neighborhood of the candidates and in the storage of the result.

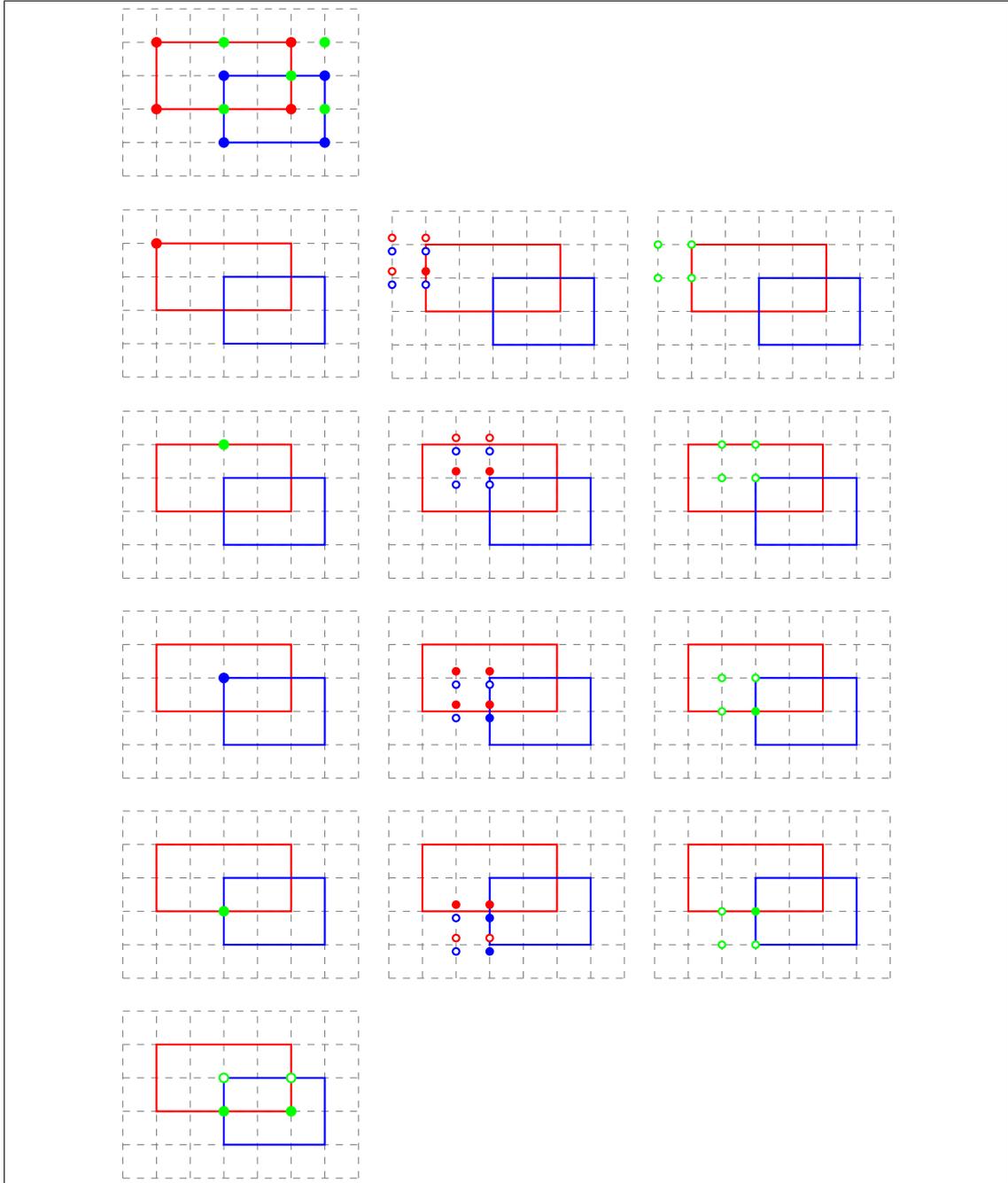


Figure 7.26: Example intersection computation

7.2 Convex Polyhedra

After orthogonal polyhedra next we discuss state set representation by *convex polyhedra*. Some polyhedra are depicted in Figure 7.27

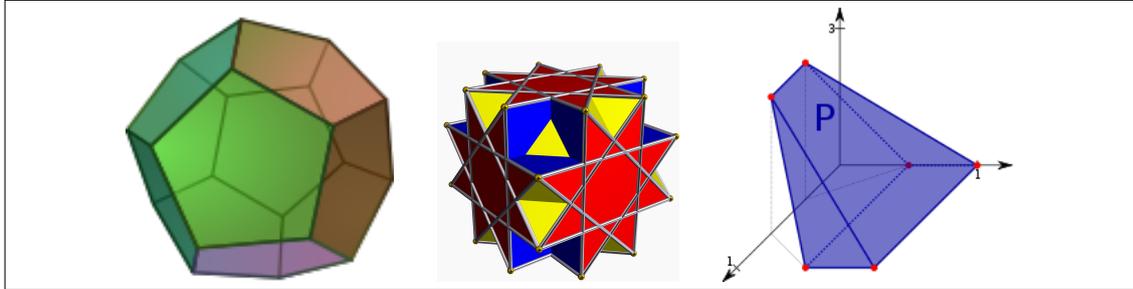


Figure 7.27: Polyhedra

Definition 58.

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 vertices¹.

Definition 59 (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 60 (\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

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

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 i th facet of the polytope. Note that each \mathcal{H} -polytope P has a finite number of vertices which we denote by $V(P)$.

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

Definition 61.

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.

Definition 62 (Convex hull).

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

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

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

Definition 63 (\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 programmms.

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

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

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

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

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

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

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 $C_1x \leq z_1$ and $C_2x \leq z_2$, 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 $C_1x \leq z_1$ and $C_2x \leq z_2$ 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

- τ -transition, 17, 19, 46
- abstraction, 38
 - timed automaton, 53–58
- activity, 22, 45
 - \sim of a rectangular automaton, 64
- always operator, *see* globally operator
- atomic proposition, 31, 47
- atomic propositions, 13
- automaton
 - hybrid, *see* hybrid automaton
 - linear hybrid, *see* linear hybrid automaton
 - rectangular, *see* rectangular automaton
 - singular, *see* singular automaton
 - stopwatch, *see* stopwatch automaton
 - rectangular automaton, 65
 - controlled variable, 17, 19, 24, 46
 - convex, 45, 46, 65
- CTL, 34–36
 - semantics, 35
 - syntax, 35
- CTL*, 36–37
 - semantics, 36
 - syntax, 36
- deterministic jump, 67
- differential equation, 23
- discrete system, 9
- discrete transition, 12, 17, 21, 22, 45, 46, 48
 - linear hybrid automaton, 70
 - rectangular automaton, 65
- discrete-time system, 40–42
- eventually operator, *see* finally operator
- example
 - railroad crossing, 48
- examples
 - bouncing ball, 11, 27
 - pedestrian light, 13, 14
 - railroad crossing, 15, 29
 - thermostat, 9, 26
 - water-level monitor, 10, 26
 - while program, 18
- execution, *see* path
- finally operator, 33, 35, 36, 40
- finite-state system, 9, 38
- forward time closure, 72
- globally operator, 33, 35, 36, 40

- graphical representation
 - hybrid automaton, 23
 - labeled state transition system, 13
 - labeled transition system, 18
 - timed automaton, 47
- HA, *see* hybrid automaton
- hybrid automaton, 21–30
 - activity, 22
 - controlled variable, 24
 - invariant, 22
 - parallel composition, 28
 - path, 23
 - reachability problem, 43
 - reachable state, 23
 - semantics, 22
 - syntax, 22
- hybrid behaviour, 21
- hybrid system, 9
- induced transition system, 18, 23, 47
- infinite-state system, 38, 54
- initialized, 64
- interleaving, 19
- interval, 51
- invariant, 22, 45, 46
 - \sim of a rectangular automaton, 64
- labeled state transition system, 12–16
 - parallel composition, 13
 - path, 12
 - reachable state, 13
 - semantics, 12
 - syntax, 12
- labeled transition system, 16–21
 - controlled variable, 17, 19
 - parallel composition, 19
 - path, 17
 - reachable state, 17
 - semantics, 17
 - state, 17
 - syntax, 17
- labeling function, 13, 31, 38, 47
- linear formula, 69
- linear hybrid automaton, 69–71
 - approximation, 74–75
 - backward analysis, 73–74
 - forward analysis, 71–73
 - minimization, 75–81
 - semantics, 70
 - syntax, 69
- linear temporal logic, *see* LTL
- linear term, 69
- liveness property, 34
- location, 16
- logic
 - CTL, *see* CTL
 - CTL*, *see* CTL*
 - LTL, *see* LTL
 - propositional, 31
 - relation of LTL, CTL, and CTL*, 37
 - TCTL, *see* TCTL
 - temporal, 32–37
 - timed temporal \sim , *see* TCTL
- LSTS, *see* labeled state transition system
- LTL, 32–34
 - semantics, 34
 - syntax, 32
- LTS, *see* labeled transition system
- model checking, 38–40
 - linear hybrid automaton, 71–81
 - TCTL, 52, 62
 - timed automaton, 43
- next operator, 36, 40, 41, 52
- next time operator, 33, 35
- non-determinism, 10, 26, 64, 70
- non-zenoness, 50
 - sufficient condition, 50
- operator
 - bounded until, 51
 - finally, 33, 35, 36, 40
 - globally, 33, 35, 36, 40
 - next, 36, 40, 41, 52
 - next time, 33, 35

- until, 33, 35, 36, 40, 41, 51
- parallel composition
 - hybrid automaton, 28
 - labeled state transition system, 13
 - labeled transition system, 19
 - timed automaton, 46, 47
- path
 - hybrid automaton, 23
 - labeled state transition system, 12
 - labeled transition system, 17
 - rectangular automaton, 65
 - time-convergent, 49, 50
 - time-divergent, 49, 50
 - timed automaton, 46
 - zeno, 49, 50
- pedestrian light, 13, 14
- postcondition, 72
- property
 - liveness, 34
 - safety, 34
- propositional logic
 - semantics, 31
 - syntax, 31
- quantifier, 34, 35, 51
- railroad crossing, 15, 29, 48
- reachability problem
 - hybrid automaton, 43
 - timed automaton, 43
- reachable state
 - timed automaton, 46
- real-time system, 50
- rectangular automaton, 63–66
 - activity, 64
 - decidability, 66–68
 - initialized, 64
 - invariant, 64
 - path, 65
 - reachable state, 65
 - semantics, 65
 - syntax, 64
- rectangular set, 63
- region, 72
 - reachable, 72
 - timed automaton, 53
- region transition system, 53, 58–62
- RTS, *see* region transition system
- run, *see* path
- safety property, 34
- semantics
 - clock constraint, 44
 - clock reset, 44
 - CTL, 35
 - CTL*, 36
 - hybrid automaton, 22
 - labeled state transition system, 12
 - labeled transition system, 17
 - linear hybrid automaton, 70
 - LTL, 34
 - propositional logic, 31
 - rectangular automaton, 65
 - TCTL, 52
 - timed automaton, 46
- shared variable, 46
- singular automaton, 68
- state
 - abstract, 53
 - labeled state transition system, 12
 - labeled transition system, 17
 - reachable
 - hybrid automaton, 23
 - labeled state transition system, 13
 - labeled transition system, 17
 - rectangular automaton, 65
 - timed automaton, 46
- stopwatch, 67
- stopwatch automaton, 67
- stutter transition, *see* τ -transition
- synchronization, 19, 46
- syntax
 - clock constraint, 43
 - clock reset, 44
 - CTL, 35
 - CTL*, 36
 - hybrid automaton, 22

- labeled state transition system, 12
- labeled transition system, 17
- linear hybrid automaton, 69
- LTL, 32
- propositional logic, 31
- rectangular automaton, 64
- TCTL, 51
- timed automaton, 46
- system
 - continuous, 9
 - discrete, 9
 - discrete-time, 40–42
 - finite state, 9, 38
 - hybrid, 9
 - infinite state, 38, 54
 - real-time, 50
 - time-critical, 50
- TCTL, 50–52
 - model checking, 52, 62
 - semantics, 49, 52
 - syntax, 51
- thermostat, 9, 26
- tick, 40
- time
 - \sim lock, 49, 50
 - bound, 40
 - continuous, 43, 49
 - convergence, 49, 50
 - discrete, 40
 - divergence, 49, 50
- time delay, 44
- time deterministic, 69, 71
- time step, *see* continuous transition
- time-critical system, 50
- time-invariant, 22, 24
- timed automaton, 43–50
 - graphical representation, 47
 - model checking, 52, 62
 - parallel composition, 46, 47
 - path, 46
 - reachability problem, 43
 - reachable state, 46
 - semantics, 46
 - syntax, 46
- timed computation tree logic, *see* TCTL
- timed temporal logic, *see* TCTL
- transition, 46
 - continuous, *see* continuous transition
 - discrete, *see* discrete transition
- transition relation, 17, 45
 - \sim of a rectangular automaton, 64
 - linear hybrid automaton, 70
- transition system
 - induced, 18, 23, 47
 - labeled \sim , *see* labeled transition system
 - labeled state \sim , *see* labeled state transition system
 - region \sim , *see* region transition system
- until operator, 33, 35, 36, 40, 41, 51
 - bounded \sim , 51
- valuation, 17, 44
- variable, 17
 - clock, 45, 46
 - controlled, 17, 19, 24, 46
 - finite-slope, 68
 - shared, 46
- water-level monitor, 10, 26
- while program, 18
- zenoness, 49, 50
- zone, 63, 66
 - multirectangular, 63
 - rectangular, 63

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. 9, 69
- [ACHH92] R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In *Hybrid Systems*, volume 736 of *LNCS*, pages 209–229. Springer-Verlag, 1992. 43
- [BK08] Baier and Katoen. *Principles of Model Checking*. The MIT Press, 2008. 43
- [BMP99] Oliver Bournez, Oded Maler, and Amir Pnueli. Orthogonal polyhedra: Representation and computation. In *Proc. of Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *LNCS*, pages 46–60. Springer-Verlag, 1999. 86
- [CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 3. print edition, 2001. 42
- [Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings 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. 9
- [HKPV98] Henzinger, Kopke, Puri, and Varaiya. What's decidable about hybrid automata? *JCSS: Journal of Computer and System Sciences*, 57, 1998. 63
- [Kat99] Joost-Pieter Katoen. *Concepts, Algorithms and Tools for Model Checking*, volume 32-1 of *Arbeitsberichte der Informatik*. Friedrich-Alexander-Universität Erlangen Nürnberg, 1999. 42
- [SK03] Olaf Stursberg and Bruce H. Krogh. Efficient representation and computation of reachable sets for hybrid systems. In *Hybrid Systems: Computation and Control (HSCC'03)*, volume 2623 of *LNCS*, pages 482–497. Springer-Verlag, 2003. 86