PREFACE

These course notes are based on notes for a one-week course on discrete event and hybrid systems that I taught at Trinity College, Dublin, in July 2009. They have also served as a basis for a course on Discrete Event Systems that I have taught at TU Berlin for a number of years. The notes were produced with the help of Tom Brunsch, Behrang Monajemi Nejad, Stephanie Geist and Germano Schafaschek. Thanks to all of them! Although this represents a revised version, there are bound to be some errors. These are of course my responsibility. I would be grateful, if you could point out any error that you spot.

Jörg Raisch
raisch@control.tu-berlin.de
CONTENTS

1 Introduction
   1.1 Discrete-Event Systems
   1.2 Course Outline

2 Petri Nets
   2.1 Petri Net Graphs
   2.2 Petri Net Dynamics
   2.3 Special Classes of Petri Nets
   2.4 Analysis of Petri Nets
      2.4.1 Petri net properties
      2.4.2 The coverability tree
   2.5 Control of Petri Nets
      2.5.1 State based control – the ideal case
      2.5.2 State based control – the nonideal case

3 Timed Petri Nets
   3.1 Timed Petri Nets with Transition Delays
   3.2 Timed Event Graphs with Transition Delays
   3.3 Timed Petri Nets with Holding Times
   3.4 Timed Event Graphs with Holding Times
   3.5 The Max-Plus Algebra
      3.5.1 Introductory example
      3.5.2 Max-Plus Basics
      3.5.3 Max-plus algebra and precedence graphs
      3.5.4 Linear implicit equations in max-plus
      3.5.5 State equations in max-plus
      3.5.6 The max-plus eigenproblem
      3.5.7 Linear independence of eigenvectors
      3.5.8 Cyclicity

4 Supervisory Control
   4.1 SCT Basics
   4.2 Plant Model
   4.3 Plant Controller Interaction
   4.4 Specifications
   4.5 Controller Realisation
      4.5.1 Finite automata with marked states
      4.5.2 Unary operations on automata
      4.5.3 Binary operations on automata
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.4</td>
<td>Realising least restrictive implementable control</td>
<td>83</td>
</tr>
<tr>
<td>4.6</td>
<td>Control of a Manufacturing Cell</td>
<td>85</td>
</tr>
</tbody>
</table>
INTRODUCTION

1.1 DISCRETE-EVENT SYSTEMS

In “conventional” systems and control theory, signals “live” in $\mathbb{R}^n$ (or some other, possibly infinite-dimensional, vector space). Then, a signal is a map $T \rightarrow \mathbb{R}^n$, where $T$ represents continuous or discrete time. There are, however, numerous application domains where signals only take values in a discrete set, which is often finite and not endowed with mathematical structure. Examples are pedestrian lights (possible signal values are “red” and “green”) or the qualitative state of a machine (“busy”, “idle”, “down”). Sometimes, such discrete-valued signals are the result of a quantisation process.

Example 1.1 Consider a water reservoir, where $y : \mathbb{R}^+ \rightarrow \mathbb{R}^+$ is the (continuous-valued) signal representing the water level in the reservoir. The quantised signal

$$y_d : \mathbb{R}^+ \rightarrow \{\text{Hi}, \text{Med}, \text{Lo}\},$$

where

$$y_d(t) = \begin{cases} \text{Hi} & \text{if } y(t) > 2 \\ \text{Med} & \text{if } 1 < y(t) \leq 2 \\ \text{Lo} & \text{if } y(t) \leq 1 \end{cases}$$

represents coarser, but often adequate, information on the temporal evolution of the water level within the reservoir. This is indicated in Fig. 1.1 which also shows that the discrete-valued signal $y_d : \mathbb{R}^+ \rightarrow \{\text{Hi}, \text{Med}, \text{Lo}\}$ can be represented by a sequence of timed discrete events, e.g.

$$(\text{LoMed}, t_1), (\text{MedHi}, t_2), (\text{HiMed}, t_3), \ldots,$$

where $t_i \in \mathbb{R}^+$ are event times and the symbol $\text{LoMed}$ denotes the event that the value of the signal $y_d$ changes from Lo to Med. Similarly, the symbols $\text{MedHi}$ and $\text{HiMed}$ represent the events that $y_d$ changes from Med to Hi and from Hi to Med, respectively. Note that a sequence of timed discrete events can be interpreted as a map $\mathbb{N} \rightarrow \mathbb{R}^+ \times \Sigma$, where $\Sigma$ is the event set. 

$\diamondsuit$
Sometimes, even less information may be required. For example, only the temporal ordering of events, but not the precise time of the occurrence of events may be relevant. In this case, the signal reduces to a sequence of logical events, e.g.

$$\text{LoMed, MedHi, HiMed, \ldots,}$$

which can be interpreted as a map $\mathbb{N} \rightarrow \Sigma$, where $\Sigma$ is the event set.

Clearly, going from the continuous-valued signal $y$ to the discrete-valued signal $y_d$ (or the corresponding sequence of timed discrete events), and from the latter to a sequence of logical events, involves a loss of information. This is often referred to as signal aggregation or abstraction.

If a dynamical system can be completely described by discrete-valued signals, or sequences of discrete events, it is said to be a discrete-event system (DES). If time is included explicitly, it is a timed DES, otherwise an untimed, or logical, DES. If a system consists of interacting DES and continuous modules, it is said to be a hybrid system.

### 1.2 Course outline

This course is organised as follows. In Chapter 2, we start with Petri nets, a special class of DES that has been popular since its inception by C.A. Petri in the 1960s. We will treat modelling and analysis aspects and discuss elementary feedback control.
problems for Petri nets. It will become clear that under some – unfortunately quite restrictive – conditions, certain optimal feedback problems can be solved very elegantly in a Petri net framework. For general Petri nets, only suboptimal solutions are available, and the solution procedure is much more involved. Then, in Chapter 3 we will investigate timed Petri nets and discuss that a subclass, the so-called timed event graphs, can be elegantly described in a max-plus algebraic framework. The max-plus algebra is an idempotent semiring and provides powerful tools for both the analysis and synthesis of timed event graphs. In Chapter 4 we will discuss the basic aspects of supervisory control theory (SCT). SCT was developed to a large extent by W.M. Wonham and coworkers. In this framework, the DES problem is modelled in a formal language scenario, and computational aspects are treated on the realisation (i.e. finite state machine) level.
1 Introduction
PETRI NETS

Petri nets provide an intuitive way of modelling discrete-event systems where “counting”, i.e., the natural numbers, play a central role. This is illustrated in the following introductory example.

Example 2.1 Two adjacent rooms in a building are connected by a door. Room B is initially empty, while there are three desks and four chairs in room A. Two people, initially also in room A, are required to carry all desks and chairs from room A to room B. While a desk can only be moved by two people, one person is sufficient to carry a chair. To describe this process, we define three events: “a desk is moved from room A to room B”, “a chair is moved from room A to room B”, and “a person walks back from room B to room A”. Furthermore, we need to keep track of the number of desks, chairs and people in each room. To do this, we introduce six counters. Counters and events are connected as shown in Fig. 2.1. The figure is to be interpreted as follows: an event can only occur if all its “upstream” counters contain at least the required number of “tokens”. For example, the event “a desk is moved from room A to room B” can only occur if there is at
least one desk left in room A and if there are (at least) two people in room A. If the event occurs, the respective “upstream” counters are decreased, and the “downstream” counters increased. In the example, the event “a desk is moved from room A to room B” obviously decreases the number of desks in room A by one, the number of people in room A by two, and increases the respective numbers for room B.

It will be pointed out in the sequel that the result is indeed a (simple) Petri net.

2.1 PETRI NET GRAPHS

Recall that a bipartite graph is a graph where the set of nodes is partitioned into two sets. In the Petri net case, the elements of these sets are called “places” and “transitions”.

**Definition 2.1 (Petri net graph)** A Petri net graph is a directed bipartite graph

\[ N = (P, T, E, w) \]

where \( P = \{p_1, \ldots, p_n\} \) is the (finite) set of places, \( T = \{t_1, \ldots, t_m\} \) is the (finite) set of transitions, \( E \subseteq (P \times T) \cup (T \times P) \) is the set of directed arcs from places to transitions and from transitions to places, and \( w : E \to \mathbb{N} \) is a weight function.

The following notation is standard for Petri net graphs:

\[ I(t_i) := \{p_i \in P \mid (p_i, t_i) \in E\} \quad (2.1) \]

is the set of all input places for transition \( t_i \), i.e., the set of places with arcs to \( t_i \).

\[ O(t_i) := \{p_i \in P \mid (t_i, p_i) \in E\} \quad (2.2) \]

denotes the set of all output places for transition \( t_i \), i.e., the set of places with arcs from \( t_i \). Similarly,

\[ I(p_i) := \{t_j \in T \mid (t_j, p_i) \in E\} \quad (2.3) \]

is the set of all input transitions for place \( p_i \), i.e., the set of transitions with arcs to \( p_i \), and

\[ O(p_i) := \{t_j \in T \mid (p_i, t_j) \in E\} \quad (2.4) \]

denotes the set of all output transitions for place \( p_i \), i.e., the set of transitions with arcs from \( p_i \). Obviously, \( p_i \in I(t_i) \) if and only if \( t_i \in O(p_i) \), and \( t_i \in I(p_i) \) if and only if \( p_i \in O(t_i) \).

In graphical representations, places are shown as circles, transitions as bars, and arcs as arrows. The number attached to an arrow is the weight of the corresponding arc. Usually, weights are only shown explicitly if they are different from one.
Example 2.2 Figure 2.2 depicts a Petri net graph with 4 places and 5 transitions. All arcs with the exception of \((p_2, t_3)\) have weight 1.

\[
\begin{array}{c}
\text{Figure 2.2: Petri net graph.}
\end{array}
\]

Remark 2.1 Often, the weight function is defined as a map

\[
w : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}_0 = \{0, 1, 2, \ldots\}.
\]

Then, the set of arcs is determined by the weight function as

\[
E = \{(p_i, t_j) \mid w(p_i, t_j) \geq 1\} \cup \{(t_j, p_i) \mid w(t_j, p_i) \geq 1\}.
\]

2.2 PETRI NET DYNAMICS

Definition 2.2 (Petri net) A Petri net is a pair \((N, x^0)\) where \(N = (P, T, E, w)\) is a Petri net graph and \(x^0 \in \mathbb{N}_0^n\), \(n = |P|\), is a vector of initial markings.

In graphical illustrations, the vector of initial markings is shown by drawing \(x^0_i\) dots (“tokens”) within the circles representing the places \(p_i\), \(i = 1, \ldots, n\).

A Petri net \((N, x^0)\) can be interpreted as a dynamical system with state signal \(x : \mathbb{N}_0 \rightarrow \mathbb{N}_0^n\) and initial state \(x(0) = x^0\). The dynamics of the system is defined by two rules:

1. in state \(x(k)\) a transition \(t_j\) can occur if and only if all of its input places contain at least as many tokens as the weight

\[1\] In the Petri net terminology, one often says “a transition can fire.”
of the arc from the respective place to the transition \( t_j \), i.e., if
\[
x_i(k) \geq w(p_i, t_j) \quad \forall p_i \in I(t_j).
\] (2.5)

2. If a transition \( t_j \) occurs, the number of tokens in all its input places is decreased by the weight of the arc connecting the respective place to the transition \( t_j \), and the number of tokens in all its output places is increased by the weight of the arc connecting \( t_j \) to the respective place, i.e.,
\[
x_i(k + 1) = \begin{cases} 
  x_i(k) - w(p_i, t_j) + w(t_j, p_i) & \text{if } p_i \in I(t_j) \cap O(t_j), \\
  x_i(k) - w(p_i, t_j) & \text{if } p_i \in I(t_j) \setminus O(t_j), \\
  x_i(k) + w(t_j, p_i) & \text{if } p_i \in O(t_j) \setminus I(t_j), \\
  x_i(k) & \text{else},
\end{cases}
\] (2.6)

where \( x_i(k) \) and \( x_i(k + 1) \) represent the numbers of tokens in place \( p_i \) before and after the firing of transition \( t_j \).

Note that a place can simultaneously be an element of \( I(t_j) \) and \( O(t_j) \). Hence the number of tokens in a certain place can appear in the firing condition for a transition whilst being unaffected by the actual firing. It should also be noted that the fact that a transition may fire (i.e., is enabled) does not imply it will actually do so. In fact, it is well possible that in a certain state several transitions are enabled simultaneously, and that the firing of one of them will disable the other ones.

The two rules stated above define the (partial) transition function \( f : \mathbb{N}_0^n \times T \to \mathbb{N}_0^n \) for the Petri net \((N, x^0)\) and hence completely describe the dynamics of the Petri net. We can therefore compute all possible evolutions of the state \( x \) starting in \( x(0) = x^0 \). This is illustrated in the following example.

**Example 2.3** Consider the Petri net graph in Fig. 2.3 with \( x^0 = (2, 0, 0, 1)' \).

Clearly, in state \( x^0 \), transition \( t_1 \) may occur, but transitions \( t_2 \) or \( t_3 \) are disabled. If \( t_1 \) fires, the state will change to \( x^1 = (1, 1, 1, 1)' \). In other words; \( f(x^0, t_1) = x^1 \) while \( f(x^0, t_2) \) and \( f(x^0, t_3) \) are undefined. If the system is in state \( x^1 \) (Fig. 2.4), all three transitions may occur and
\[
\begin{align*}
  f(x^1, t_1) & = (0, 2, 2, 1)' =: x^2 \\
  f(x^1, t_2) & = (1, 1, 0, 2)' =: x^3 \\
  f(x^1, t_3) & = (0, 1, 0, 0)' =: x^4 \\
\end{align*}
\]

It can be easily checked that \( f(x^4, t_j) \) is undefined for all three
transitions, i.e., the state $x^4$ represents a deadlock, and that
$$f(x^2, t_2) = f(x^3, t_1) = (0, 2, 1, 2)' =: x^5,$$
while $f(x^2, t_1), f(x^2, t_3), f(x^3, t_2),$ and $f(x^3, t_3)$ are all undefined. Finally, in $x^5$, only transition $t_2$ can occur, and this will lead into another deadlock $x^6 := f(x^5, t_2)$. The evolution of the state can be conveniently represented as a reachability graph (Fig. 2.5).
To check whether a transition can fire in a given state and, if the answer is affirmative, to determine the next state, it is convenient to introduce the matrices $A^-, A^+ \in \mathbb{N}_0^{n \times m}$ by 

$$a^-_{ij} = \begin{cases} w(p_i, t_j) & \text{if } (p_i, t_j) \in E \\ 0 & \text{otherwise} \end{cases} \quad \text{(2.7)}$$

$$a^+_{ij} = \begin{cases} w(t_j, p_i) & \text{if } (t_j, p_i) \in E \\ 0 & \text{otherwise} \end{cases} \quad \text{(2.8)}$$

The matrix $A := A^+ - A^- \in \mathbb{Z}^{n \times m}$ (2.9) is called the incidence matrix of the Petri net graph $N$. Clearly, $a^-_{ij}$ represents the number of tokens that place $p_i$ loses when transition $t_j$ fires, and $a^+_{ij}$ is the number of tokens that place $p_i$ gains when transition $t_j$ fires. Consequently, $a_{ij}$ is the net gain (or loss) for place $p_i$ when transition $t_j$ occurs. We can now rephrase (2.5) and (2.6) as follows:

1. The transition $t_j$ can fire in state $x(k)$ if and only if 
   
   $$x(k) \geq A^- u_j \quad \text{(2.10)}$$

   where the “≥”-sign is to be interpreted elementwise and where $u_j$ is the $j$-th unit vector in $\mathbb{Z}^m$.

2. If transition $t_j$ fires, the state changes according to 
   
   $$x(k+1) = x(k) + A u_j \quad \text{(2.11)}$$

**Remark 2.2** Up to now, we have identified the firing of transitions and the occurrence of events. Sometimes, it may be useful to distinguish transitions and events, for example, when different transitions are associated with the same event. To do this, we simply introduce a (finite) event set $F$ and define a surjective map $\lambda : T \to F$ that associates an event in $F$ to every transition $t_j \in T$.

We close this section with two more examples to illustrate how Petri nets model certain discrete event systems.

**Example 2.4** This example is taken from [3]. We consider a simple queueing system with three events (transitions):

- $a \ldots$ “customer arrives”,
- $s \ldots$ “service starts”,
- $c \ldots$ “service complete and customer departs”.

Clearly, the event $a$ corresponds to an autonomous transition, i.e., a transition without input places. If we assume that only one
2.2 Petri Net Dynamics

Figure 2.6: Petri net model for queueing system.

customer can be served at any instant of time, the behaviour of the queueing system can be modelled by the Petri net shown in Fig. 2.6. For this Petri net, the matrices \( A^- \), \( A^+ \) and \( A \) are given by:

\[
A^- = \begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 1 \\
0 & 1 & 0
\end{bmatrix}
\]

\[
A^+ = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}
\]

\[
A = \begin{bmatrix}
1 & -1 & 0 \\
0 & 1 & -1 \\
0 & -1 & 1
\end{bmatrix}
\]

Example 2.5 We now model a candy machine. It sells three products: “Mars” (for 80 Cents), “Bounty” (for 70 Cents) and “Milky Way” (for 40 Cents). The machine accepts only the following coins: 5 Cents, 10 Cents, 20 Cents and 50 Cents. Finally, change is only given in 10 Cents coins. The machine is supposed to operate in the following way: the customer inserts coins and requests a product; if (s)he has paid a sufficient amount of money and the product is available, it is given to the customer. If (s)he has paid more than the required amount and requests change, and if 10 Cents coins are available, change will be given. This can be modelled by the Petri net shown in Fig. 2.7.
2 Petri Nets

![Petri Net Diagram]

Figure 2.7: Petri net model for candy machine.

2.3 SPECIAL CLASSES OF PETRI NETS

There are two important special classes of Petri nets.

**Definition 2.3 (Event graph)** A Petri net $(N, x^0)$ is called an event graph (or synchronisation graph), if each place has exactly one input transition and one output transition, i.e.

$$|I(p_i)| = |O(p_i)| = 1 \quad \forall p_i \in P,$$

and if all arcs have weight 1, i.e.

$$w(p_i, t_j) = 1 \quad \forall (p_i, t_j) \in E$$

$$w(t_j, p_i) = 1 \quad \forall (t_j, p_i) \in E.$$ 

**Definition 2.4 (State machine)** A Petri net $(N, x^0)$ is called a state machine, if each transition has exactly one input place and one output place, i.e.

$$|I(t_j)| = |O(t_j)| = 1 \quad \forall t_j \in T,$$

and if all arcs have weight 1, i.e.

$$w(p_i, t_j) = 1 \quad \forall (p_i, t_j) \in E$$

$$w(t_j, p_i) = 1 \quad \forall (t_j, p_i) \in E.$$ 

Figs. 2.8 and 2.9 provide examples for an event graph and a state machine, respectively. It is obvious that an event graph cannot model conflicts or decisions but it does model synchronisation.

---

2 For this reason, event graphs are sometimes also called decision free Petri nets.
2.4 Analysis of Petri Nets

effects. A state machine, on the other hand, can model conflicts but does not describe synchronisation effects.

![Event graph example](image1)

Figure 2.8: Event graph example.

![State machine example](image2)

Figure 2.9: State machine example.

2.4 ANALYSIS OF PETRI NETS

In this section, we define a number of important properties for Petri nets. Checking if these properties hold is in general a nontrivial task, as the state set of a Petri net may be infinite. Clearly, in such a case, enumeration-type methods will not work. For this reason, the important concept of a coverability tree has become popular in the Petri net community. It is a finite entity and can be used to state conditions (not always necessary and sufficient) for most of the properties discussed next.
2.4.1 Petri net properties

It will be convenient to work with the Kleene closure $T^*$ of the transition set $T$. This is the set of all finite strings of elements from $T$, including the empty string $\epsilon$. We can then extend the (partial) transition function $f : \mathbb{N}_0^n \times T \rightarrow \mathbb{N}_0^n$ to $f : \mathbb{N}_0^n \times T^* \rightarrow \mathbb{N}_0^n$ in a recursive fashion:

$$
f(x^0, \epsilon) = x^0$$
$$f(x^0, st_j) = f(f(x^0, s), t_j) \quad \text{for } s \in T^* \text{ and } t_j \in T,$$

where $st_j$ is the concatenation of $s$ and $t_j$, i.e., the string $s$ followed by the transition $t_j$.

**Definition 2.5 (Reachability)** A state $x^l \in \mathbb{N}_0^n$ of the Petri net $(N, x^0)$ is said to be reachable, if there is a string $s \in T^*$ such that $x^l = f(x^0, s)$. The set of reachable states of the Petri net $(N, x^0)$ is denoted by $R(N, x^0)$.

**Definition 2.6 (Boundedness)** A place $p_i \in P$ is bounded, if there exists a $k \in \mathbb{N}_0$ such that $x^l_i \leq k$ for all $x^l \in R(N, x^0)$. The Petri net $(N, x^0)$ is bounded if all its places are bounded.

It is obvious that a Petri net is bounded if and only if its reachable set is finite.

**Example 2.6** Consider the Petri net in Fig. 2.10. It is clearly unbounded as transition $t_1$ can fire arbitrarily often, and each firing of $t_1$ consumes less tokens than it generates.

---

2 Petri Nets
The next property we discuss is related to the question whether we can reach a state \( x^l \) where the transition \( t_j \in T \) can fire. As discussed earlier, \( t_j \) can fire in state \( x^l \), if \( x^l_i \geq w(p_i, t_j) \ \forall p_j \in I(t_j) \) or, equivalently, if

\[
x^l \geq A^- u_j := \xi^l
\]

(2.12)

where the “\( \geq \)”-sign is to be interpreted elementwise. If (2.12) holds, we say that \( x^l \) covers \( \xi^l \). This is captured in the following definition.

**Definition 2.7 (Coverability)** The vector \( \xi \in \mathbb{N}^n \) is coverable if there exists an \( x^l \in R(N, x^0) \) such that \( x^l_i \geq \xi_i, i = 1, \ldots, n \).

**Example 2.7** Consider the Petri net shown in the left part of Fig. 2.11.

Clearly,

\[
A^- = \begin{bmatrix}
1 & 1 & 1 \\
0 & 1 & 0
\end{bmatrix}.
\]

Hence, to enable transition \( t_2 \), it is necessary for the state \( \xi^2 = A^- u_2 = (1, 1)' \) to be coverable. In other words, a state in the shaded area in the right part of Fig. 2.11 needs to be reachable. This is not possible, as the set of reachable states consists of only two elements, \( x^0 = (1, 0)' \) and \( x^1 = (0, 1)' \).

**Definition 2.8 (Conservation)** The Petri net \( (N, x^0) \) is said to be conservative with respect to \( \gamma \in \mathbb{Z}^n \) if

\[
\gamma' x^l = \sum_{j=1}^n \gamma_i x^l_j = \text{const.} \ \forall x^l \in R(N, x^0) .
\]

(2.13)

The interpretation of this property is straightforward. As the system state \( x(k) \) will evolve within the reachable set, it will also be restricted to the hyperplane (2.13).

**Example 2.8** Consider the queueing system from Example 2.4. The Petri net shown in Fig. 2.6 is conservative with respect to \( \gamma = (0, 1, 1)' \), and its state \( x \) will evolve on the hyperplane shown in Fig. 2.12.

\[\Box\]
Definition 2.9 (Liveness) A transition \( t_j \in T \) of the Petri net \((N, x^0)\) is said to be

- **dead**, if it can never fire, i.e., if the vector \( \xi^j = A^- u_j \) is not coverable by \((N, x^0)\),

- **L1-live**, if it can fire at least once, i.e., if \( \xi^j = A^- u_j \) is coverable by \((N, x^0)\),

- **L3-live**, if it can fire arbitrarily often, i.e., if there exists a string \( s \in T^* \) that contains \( t_j \) arbitrarily often and for which \( f(x^0, s) \) is defined,

- **live**, if, from any reachable state, it is possible to reach a state where \( t_j \) can fire, i.e., if \( \xi^j = A^- u_j \) can be covered by \((N, x^i) \forall x^i \in R(N, x^0)\).

Example 2.9 Consider the Petri net from Example 2.7. Clearly, \( t_1 \) is L1-live (but not L3-live), transition \( t_2 \) is dead, and \( t_3 \) is L3-live, but not live. The latter is obvious, as \( t_3 \) may fire arbitrarily often, but will be permanently disabled by the firing of \( t_1 \).

Definition 2.10 (Persistence) A Petri net \((N, x^0)\) is persistent, if, for any pair of simultaneously enabled transitions \( t_{j_1}, t_{j_2} \in T \), the firing of \( t_{j_1} \) will not disable \( t_{j_2} \).

Example 2.10 The Petri net from Example 2.7 is not persistent: in state \( x^0 \), both transitions \( t_1 \) and \( t_3 \) are enabled simultaneously, but the firing of \( t_1 \) will disable \( t_3 \).
2.4 Analysis of Petri Nets

2.4.2 The coverability tree

We start with the reachability graph of the Petri net \((N, x^0)\). In Fig. 2.5, we have already seen a specific example for this. The nodes of the reachability graph are the reachable states of the Petri net, the edges are the transitions that are enabled in these states.

A different way of representing the reachable states of a Petri net \((N, x^0)\) is the reachability tree. This is constructed as follows: one starts with the root node \(x^0\). We then draw arcs for all transitions \(t_j \in T\) that can fire in the root node and draw the states \(x^i = f(x^0, t_j)\) as successor nodes. In each of the successor states we repeat the process. If we encounter a state that is already a node in the reachability tree, we stop.

Clearly, the reachability graph and the reachability tree of a Petri net will only be finite, if the set of reachable states is finite.

Example 2.11 Consider the Petri net shown in Fig. 2.13 (taken from [3]). Apart from the initial state \(x^0 = (1, 1, 0)\) only the state \(x^1 = (0, 0, 1)\) is reachable. Hence both the reachability graph (shown in the left part of Fig. 2.14) and the reachability tree (shown in the right part of Fig. 2.14) are trivial.

Unlike the reachability tree, the coverability tree of a Petri net \((N, x^0)\) is finite even if its reachable state set is infinite. The underlying idea is straightforward: if a place is unbounded, it is labelled with the symbol \(\omega\). This can be thought of as “infinity”, therefore the symbol \(\omega\) is defined to be invariant under the addition (or subtraction) of integers, i.e.,

\[ \omega + k = \omega \quad \forall k \in \mathbb{Z} \]
and

\[ \omega > k \ \forall k \in \mathbb{Z} . \]

The construction rules for the coverability tree are given below:

1. Start with the root node \( x^0 \). Label it as “new”.

2. For each new node \( x^k \), evaluate \( f(x^k, t_j) \) for all \( t_j \in T \).
   a) If \( f(x^k, t_j) \) is undefined for all \( t_j \in T \), the node \( x^k \) is a terminal node (deadlock).
   b) If \( f(x^k, t_j) \) is defined for some \( t_j \), create a new node \( x^l \).
      i. If \( x^k_i = \omega \), set \( x^l_i = \omega \).
      ii. Examine the path from the root node to \( x^k \). If there exists a node \( \xi \) in this path which is covered by, but not equal to, \( f(x^k, t_j) \), set \( x^l_i = \omega \) for all \( i \) such that \( f_i(x^k, t_j) > \xi_i \).
      iii. Otherwise, set \( x^l_i = f_i(x^k, t_j) \).
   c) Label \( x^k \) as “old”.

3. If all new nodes are terminal nodes or duplicates of existing nodes, stop.

**Example 2.12** This example is taken from [3]. We investigate the Petri net shown in Fig. 2.15. It has an infinite set of reachable states, hence its reachability tree is also infinite. We now determine the coverability tree. According to the construction rules, the root node is \( x^0 = (1, 0, 0, 0)' \). The only transition enabled in this state is \( t_1 \). Hence, we have to create one new node \( x^1 \). We now examine the rules 2.a)i.–iii. to determine the elements of \( x^1 \): as its predecessor node \( x^0 \) does not contain any \( \omega \)-symbol, rule i. does not apply. For rule ii., we investigate the path from the root node to the predecessor node \( x^0 \). This is trivial, as the path only consists of the root node itself. As the root node is not covered by \( f(x^0, t_1) = (0, 1, 1, 0)' \), rule ii. does also not apply, and therefore, according to rule iii., \( x^1 = f(x^0, t_1) = (0, 1, 1, 0)' \) (see Fig. 2.16).
In node $x^1$, transitions $t_2$ and $t_3$ are enabled. Hence, we have to generate two new nodes, $x^2$, corresponding to $f(x^1, t_2)$, and $x^3$, corresponding to $f(x^1, t_3)$. For $x^2$, rule ii. applies, as the path from the root node $x^0$ to the predecessor node $x^1$ contains a node $\xi$ that is covered by, but is not equal to, $f(x^1, t_2) = (1,0,1,0)'$. This is the root node itself, i.e., $\xi = (1,0,0,0)'$. We therefore set $x^2_3 = \omega$. For the other elements in $x^2$ we have according to rule iii. $x^2_i = f_i(x^1, t_2)$, $i = 1, 2, 4$. Hence, $x^2 = (1,0,\omega,0)'$. For $x^3$ neither rule i., nor ii. applies. Therefore, according to rule iii., $x^3 = f(x^1, t_3) = (0,0,1,1)$.

In node $x^2$, only transition $t_1$ may fire, and we have to create one new node, $x^4$. Now, rule i. applies, and we set $x^4_3 = \omega$. Rule ii. also applies, but this provides the same information, i.e., $x^4_3 = \omega$. The other elements of $x^4$ are determined according to rule iii., therefore $x^4 = (0,1,\omega,0)'$. In node $x^3$, no transition is enabled – this node represents a deadlock and is therefore a terminal node.

By the same reasoning, we determine two successor nodes for $x^4$, namely $x^5 = (1,0,\omega,0)'$ and $x^6 = (0,0,\omega,1)'$. The former is a duplicate of $x^2$, and $x^6$ is a deadlock. Therefore, the construction is finished.

\[ \diamond \]
Let $s = t_1 \ldots t_N$ be a string of transitions from $T$. We say that $s$ is compatible with the coverability tree, if there exist nodes $x^1, \ldots, x^{N+1}$ such that $x^1$ is the root node and $x^j \xrightarrow{t_i} x^{j+1}$ are transitions in the tree, $j = 1, \ldots, N$. Note that duplicate nodes are considered to be identical, hence the string $s$ can contain more transitions than there are nodes in the coverability tree.

**Example 2.13** In Example 2.12, the string $s = t_1 t_2 t_1 t_2 t_1 t_2 t_1$ is compatible with the coverability tree.

The coverability tree has a number of properties which make it a convenient tool for analysis:

1. The coverability tree of a Petri net $(N, x^0)$ with a finite number of places and transitions is finite.

2. If $f(x^0, s)$, $s \in T^*$, is defined for the Petri net $(N, x^0)$, the string $s$ is also compatible with the coverability tree.

3. The Petri net state $x^i = f(x^0, s)$, $s \in T^*$, is covered by the node in the coverability tree that is reached from the root node via the string $s$ of transitions.

The converse of item 2. above does not hold in general. This is illustrated by the following example.

**Example 2.14** Consider the Petri net in the left part of Fig. 2.17. Its coverability tree is shown in the right part of the same figure.

![Figure 2.17: Counter example.](image)

Clearly, a string of transitions beginning with $t_1 t_2 t_1$ is not possible for the Petri net, while it is compatible with the coverability tree.

The following statements follow from the construction and the properties of the coverability tree discussed above:

**Reachability:** A necessary condition for $\xi$ to be reachable in $(N, x^0)$ is that there exists a node $x^k$ in the coverability tree such that $\xi_i \leq x^k_i$, $i = 1, \ldots, n$. 

---

26
2.5 Control of Petri Nets

**Boundedness:** A place \( p_i \in P \) of the Petri net \((N, x^0)\) is bounded if and only if \( x^k_i \neq \omega \) for all nodes \( x^k \) of the coverability tree. The Petri net \((N, x^0)\) is bounded if and only if the symbol \( \omega \) does not appear in any node of its coverability tree.

**Coverability:** The vector \( \xi \) is coverable by the Petri net \((N, x^0)\) if and only if there exists a node \( x^k \) in the coverability tree such that \( \xi_i \leq x^k_i, i = 1, \ldots, n \).

**Conservation:** A necessary condition for \((N, x^0)\) to be conservative with respect to \( \gamma \in \mathbb{N}^n_0 \) is that \( \gamma_i = 0 \) if there exists a node \( x^k \) in the coverability tree with \( x^k_i = \omega \). If, in addition, \( \gamma' x^k = \text{const.} \) for all nodes \( x^k \) in the coverability tree, the Petri net is conservative with respect to \( \gamma \). Note that this condition does not hold for the more general case when \( \gamma \in \mathbb{Z}^n \).

**Dead Transitions:** A transition \( t_j \) of the Petri net \((N, x^0)\) is dead if and only if no edge in the coverability tree is labelled by \( t_j \).

However, on the basis of the coverability tree we cannot decide about liveness of transitions or the persistence of the Petri net \((N, x^0)\). This is again illustrated by a simple example:

**Example 2.15** Consider the Petri nets in Figure 2.18. They have the same coverability tree (shown in Fig. 2.17). For the Petri net shown in the left part of Fig. 2.18, transition \( t_1 \) is not live, and the net is not persistent. For the Petri net shown in the right part of the figure, \( t_1 \) is live, and the net is persistent.

![Figure 2.18: Counter example.](image)

2.5 CONTROL OF PETRI NETS

We start the investigation of control topics for Petri nets with a simple example.

**Example 2.16** Suppose that the plant to be controlled is modelled by the Petri net \((N, x^0)\) shown in Fig. 2.19. Suppose furthermore
that we want to make sure that the following inequality holds for the plant state $x$ at all times $k$:

$$x_2(k) + 3x_4(k) \leq 3 , \quad (2.14)$$

i.e., we want to restrict the plant state to a subset of $\mathbb{N}_0^4$. Without control the specification (2.14) cannot be guaranteed to hold as there are reachable states violating this inequality. However, it is easy to see how we can modify $(N, x^0)$ appropriately. Intuitively, the problem is the following: $t_1$ can fire arbitrarily often, with the corresponding number of tokens being deposited in place $p_2$. If subsequently $t_2$ and $t_4$ fire, we will have a token in place $p_4$, while there are still a large number of tokens in place $p_2$. Hence the specification will be violated. To avoid this, we add restrictions for the firing of transitions $t_1$ and $t_4$. This is done by introducing an additional place, $p_c$, with initially three tokens. It is connected to $t_1$ by an arc of weight 1, and to $t_4$ by an arc of weight 3 (see Fig. 2.20). This will certainly enforce the

$$x_2(k) + 3x_4(k) \leq 3 , \quad (2.14)$$

specification (2.14), as it either allows $t_1$ to fire (three times at
the most) or \( t_4 \) (once). However, this solution is unnecessarily conservative: we can add another arc (with weight 1) from \( t_3 \) to the new place \( p_c \) to increase the number of tokens in \( p_c \) without affecting (2.14).

The number of tokens in the new place \( p_c \) can be seen as the controller state, which affects (and is affected by) the firing of the transitions in the plant Petri net \((N, x^0)\).

In the following, we will formalise the procedure indicated in the example above.

### 2.5.1 State based control – the ideal case

Assume that the plant model is given as a Petri net \((N, x^0)\), where \( N = (P, T, E, w) \) is the corresponding Petri net graph.

Assume furthermore that the aim of control is to restrict the evolution of the plant state \( x \) to a specified subset of \( \mathbb{N}^n_0 \). This subset is given by a number of linear inequalities:

\[
\gamma'_1 x(k) \leq b_1 \\
\vdots \\
\gamma'_q x(k) \leq b_q
\]

where \( \gamma_i \in \mathbb{Z}^n, b_i \in \mathbb{Z}, i = 1, \ldots, q \). This can be written more compactly as

\[
\begin{bmatrix}
\gamma'_1 \\
\vdots \\
\gamma'_q
\end{bmatrix}
x(k) \leq 
\begin{bmatrix}
b_1 \\
\vdots \\
b_q
\end{bmatrix},
\]

(2.15)

where \( \Gamma \in \mathbb{Z}^{q \times n}, b \in \mathbb{Z}^q \), and the “\( \leq \)”-sign is to be interpreted elementwise.

The mechanism of control is to prevent the firing of certain transitions. For the time being, we assume that the controller to be synthesised can observe and – if necessary – prevent the firing of all transitions in the plant. This is clearly an idealised case. We will discuss later how to modify the control concept to handle nonobservable and/or nonpreventable transitions.

In this framework, control is implemented by creating new places \( p_{c1}, \ldots, p_{cq} \) ("controller places"). The corresponding vector of markings, \( x_c(k) \in \mathbb{N}^n_0 \), can be interpreted as the controller state. We still have to specify the initial marking of the controller places and how controller places are connected to plant transitions. To do this, consider the extended Petri net with state \((x', x'_c)')\. If
a transition $t_j$ fires, the state of the extended Petri net changes according to

$$
\begin{bmatrix}
    x(k+1) \\
    x_c(k+1)
\end{bmatrix} = \begin{bmatrix}
    x(k) \\
    x_c(k)
\end{bmatrix} + \begin{bmatrix}
    A \\
    A_c
\end{bmatrix} u_j, \quad (2.16)
$$

where $u_j$ is the $j$-th unit-vector in $\mathbb{Z}^m$ and $A_c$ is the yet unknown part of the incidence matrix. In the following, we adopt the convention that for any pair $p_{ci}$ and $t_j$, $i = 1, \ldots, q$, $j = 1, \ldots, m$, we either have an arc from $p_{ci}$ to $t_j$ or from $t_j$ to $p_{ci}$ (or no arc at all). Then, the matrix $A_c$ completely specifies the interconnection structure between controller places and plant transitions, as the non-zero entries of $A_c^+$ are the positive entries of $A_c$ and the non-zero entries of $-A_c^-$ are the negative entries of $A_c$.

To determine the yet unknown entities, $x^0_c = x_c(0)$ and $A_c$, we argue as follows: the specification $(2.15)$ holds if

$$
\Gamma x(k) + x_c(k) = b, \quad k = 0, 1, 2, \ldots \quad (2.17)
$$
or, equivalently,

$$
\begin{bmatrix}
    \Gamma & I
\end{bmatrix} \begin{bmatrix}
    x(k) \\
    x_c(k)
\end{bmatrix} = b, \quad k = 0, 1, 2, \ldots \quad (2.18)
$$
as $x_c(k)$ is a nonnegative vector of integers. For $k = 0$, Eqn. $(2.17)$ provides the vector of initial markings for the controller states:

$$
x^0_c = x_c(0) = b - \Gamma x(0) = b - \Gamma x^0. \quad (2.19)
$$

Inserting $(2.16)$ into $(2.18)$ and taking into account that $(2.18)$ also has to hold for the argument $k + 1$ results in

$$
\begin{bmatrix}
    \Gamma & I
\end{bmatrix} \begin{bmatrix}
    A \\
    A_c
\end{bmatrix} u_j = 0, \quad j = 1, \ldots, m,
$$
and therefore

$$
A_c = -\Gamma A. \quad (2.20)
$$

$(2.19)$ and $(2.20)$ solve our control problem: $(2.19)$ provides the initial value for the controller state, and $(2.20)$ provides information on how controller places and plant transitions are connected. The following important result can be easily shown.

**Theorem 2.1** $(2.19)$ and $(2.20)$ is the least restrictive, or maximally permissive, control for the Petri net $(N, x^0)$ and the specification $(2.15)$. 


Proof Recall that for the closed-loop system, by construction, \((2.17)\) holds. Now assume that the closed-loop system is in state 
\((x'(k), x'_c(k))'\), and that transition \(t_j\) is disabled, i.e.
\[
\begin{bmatrix}
x(k) \\
x_c(k)
\end{bmatrix} \geq \begin{bmatrix} A^- \\ A_c^-
\end{bmatrix} u_j
\]
does not hold. This implies that either
\begin{itemize}
  \item \(x_i(k) < (A^- u_j)_i\) for some \(i \in \{1, \ldots, n\}\), i.e., the transition is disabled in the uncontrolled Petri net \((N, x^0)\), or
  \item for some \(i \in \{1, \ldots, q\}\)
  \[
  x_c_i(k) < (A^- u_j)_i = (A^-)_{ij}
  \]
  and therefore
  \[
  x_c_i(k) < (-A_c)_{ij} = (-A_c u_j)_i = \gamma'_{ij} A u_j.
  \]
\end{itemize}

Because of \((2.17)\), \(x_c_i(k) = b_i - \gamma'_i x(k)\) and therefore
\[
  b_i < \gamma'_i (x(k) + A u_j).
\]

This means that if transition \(t_j\) could fire in state \(x(k)\) of the open-loop Petri net \((N, x^0)\), the resulting state \(x(k+1) = x(k) + A u_j\) would violate the specification \((2.15)\).

Hence, we have shown that a transition \(t_j\) will be disabled in state \((x'(k), x'_c(k))'\) of the closed-loop system if and only if it is disabled in state \(x(k)\) of the uncontrolled Petri net \((N, x^0)\) or if its firing would violate the specifications.

Example 2.17 Let’s reconsider Example 2.16 but with a slightly more general specification. We now require that
\[
x_2(k) + M x_4(k) \leq M, \quad k = 0, 1, \ldots,
\]
where \(M\) represents a positive integer. As there is only one scalar constraint, we have \(q = 1\), \(\Gamma\) is a row vector, and \(b\) is a scalar. We now apply our solution procedure for \(\Gamma = [0 \ 1 \ 0 \ M]\) and \(b = M\). We get one additional (controller) place \(p_c\) with initial marking \(x^0_c = b - \Gamma x^0 = M\). The connection structure is determined by \(A_c = -\Gamma A = [-1 \ 0 \ 1 \ -M]\), i.e., we have an arc from \(p_c\) to \(t_1\) with weight 1, an arc from \(p_c\) to \(t_4\) with weight \(M\), and an arc from \(t_3\) to \(p_c\) with weight 1. For \(M = 3\) this solution reduces to the extended Petri net shown in Fig. 2.20.

\(2.21\) implies that \((A^-)_{ij}\) is positive. Therefore, by assumption, \((A_c^+)_{ij} = 0\) and \((A_c^-)_{ij} = -(A_c)_{ij}.\)
2.5.2 State based control – the nonideal case

Up to now we have examined the ideal case where the controller could directly observe and prevent, or control, all plant transitions. It is much more realistic, however, to drop this assumption. Hence,

- a transition \( t_j \) may be uncontrollable, i.e., the controller will not be able to directly prevent the transition from firing, i.e., there will be no arc from any controller place to \( t_j \) \( \in \mathcal{T} \);

- a transition \( t_j \in \mathcal{T} \) may be unobservable, i.e., the controller will not be able to directly notice the firing of the transition. This means that the firing of \( t_j \) may not affect the number of tokens in any controller place. As we still assume that for any pair \( p_{ci} \) and \( t_j \), \( i = 1, \ldots, q \), \( j = 1, \ldots, m \), we either have an arc from \( p_{ci} \) to \( t_j \) or from \( t_j \) to \( p_{ci} \) (or no arc at all), this implies that there are no arcs from an unobservable transition \( t_j \) to any controller place or from any controller place to \( t_j \).

Then, obviously, a transition being unobservable implies that it is also uncontrollable, and controllability of a transition implies its observability. We therefore have to distinguish three different kinds of transitions: (i) controllable transitions, (ii) uncontrollable but observable transitions, and (iii) uncontrollable and unobservable transitions. We partition the set \( \mathcal{T} \) accordingly:

\[
\mathcal{T} = \mathcal{T}_{oc} \cup \mathcal{T}_{ouc} \cup \mathcal{T}_{uouc},
\]

where \( \mathcal{T}_{oc} \) and \( \mathcal{T}_{uc} \) are the sets of controllable and uncontrollable transitions, respectively. \( \mathcal{T}_{ouc} \) represents the set of uncontrollable but observable transitions, while \( \mathcal{T}_{uouc} \) contains all transitions that are both uncontrollable and unobservable.

Without loss of generality, we assume that the transitions are ordered as indicated by the partition \((2.22)\), i.e. \( t_1, \ldots, t_{m_c} \) are controllable (and observable), \( t_{m_c+1}, \ldots, t_{m_c+m_o} \) are uncontrollable but observable, and \( t_{m_c+m_o+1}, \ldots, t_m \) are uncontrollable and unobservable transitions. This implies that the incidence matrix \( A \) of the plant Petri net \((\mathcal{N}, x^0)\) has the form

\[
A = \begin{bmatrix} A_{oc} & A_{ouc} & A_{uouc} \\ \hline & A_{uc} \end{bmatrix},
\]

where the \( n \times m_c \) matrix \( A_{oc} \) corresponds to controllable (and observable) transitions etc.
**Definition 2.11 (Ideal Enforceability)** The specification (2.18) is said to be ideally enforceable, if the (ideal) controller (2.19), (2.20) can be realised, i.e., if there are no arcs from controller places to transitions in $T_{uc}$ and no arcs from transitions in $T_{uouc}$ to controller places.

Ideal enforceability is easily checked: we just need to compute the controller incidence matrix

$$A_c = -\Gamma A$$

$$= [-\Gamma A_{oc} -\Gamma A_{uouc} -\Gamma A_{wouc}]$$.

Ideal enforceability of (2.18) is then equivalent to the following three requirements:

1. $-\Gamma A_{ouc} \geq 0$ (2.23)
2. $-\Gamma A_{wouc} = 0$ (2.24)
3. $\Gamma x^0 \leq b$ (2.25)

where the inequality-signs are to be interpreted elementwise.

(2.23) says that the firing of any uncontrollable but observable transition will not depend on the number of tokens in a controller place, but may increase this number.

(2.24) means that the firing of any uncontrollable and unobservable transition will not affect the number of tokens in a controller place.

(2.25) says that there is a vector of initial controller markings that satisfies (2.18).

If a specification is ideally enforceable, the presence of uncontrollable and/or unobservable transitions does not pose any problem, as the controller (2.19), (2.20) respects the observability and controllability constraints.

If (2.18) is not ideally enforceable, the following procedure [6] can be used:

1. Find a specification

$$\Gamma x(k) \leq \bar{b} , \quad k = 0, 1, \ldots$$

which is ideally enforceable and at least as strict as (2.18). This means that $\Gamma \xi \leq \bar{b}$ implies $\Gamma \xi \leq b$ for all $\xi \in R(N, x^0)$.

2. Compute the controller (2.19), (2.20) for the new specification (2.26), i.e.

$$A_c = -\Gamma A$$

$$x^0_c = \bar{b} - \Gamma x^0$$.
Clearly, if we succeed in finding a suitable specification (2.26), the problem is solved. However, the solution will in general not be least restrictive in terms of the original specification. For the actual construction of a suitable new specification, [5] suggests the following:

Define:

\[
\begin{align*}
\overline{\Gamma} & := R_1 + R_2 \Gamma \\
\overline{b} & := R_2(b + v) - v
\end{align*}
\]

where

\[
v := (1, \ldots, 1)'
\]

\[R_1 \in \mathbb{Z}^{q \times n} \text{ such that } R_1 \xi \geq 0 \quad \forall \xi \in R(N, x^0)
\]

\[R_2 = \text{diag}(r_2) \quad \text{with } r_2 \in \mathbb{N}, \quad i = 1, \ldots, q
\]

Then, it can be easily shown that (2.26) is at least as strict as (2.18):

\[
\overline{\Gamma} \xi \leq \overline{b} \iff (R_1 + R_2 \Gamma) \xi \leq R_2(b + v) - v
\]

\[
\iff (R_1 + R_2 \Gamma) \xi < R_2(b + v)
\]

\[
\iff R_2^{-1} R_1 \xi + \Gamma \xi < b + v
\]

\[
\Rightarrow \Gamma \xi < b + v \quad \forall \xi \in R(N, x^0)
\]

\[
\Rightarrow \Gamma \xi \leq b
\]

We can now choose the entries for \(R_1\) and \(R_2\) to ensure ideal enforceability of (2.26). According to (2.23), (2.24) and (2.25), this implies

\[
(R_1 + R_2 \Gamma) A_{\text{ousc}} \leq 0
\]

\[
(R_1 + R_2 \Gamma) A_{\text{housc}} = 0
\]

\[
(R_1 + R_2 \Gamma) x^0 \leq R_2(b + v) - v
\]

or, equivalently,

\[
\begin{bmatrix} R_1 & R_2 \end{bmatrix} \begin{bmatrix} A_{\text{ousc}} & A_{\text{housc}} & -A_{\text{housc}} & \Gamma x^0 \\ \Gamma A_{\text{ousc}} & \Gamma A_{\text{housc}} & -\Gamma A_{\text{housc}} & \Gamma x^0 - b - v \end{bmatrix} \leq \begin{bmatrix} 0 & 0 & 0 & -v \end{bmatrix},
\]

where the “\(\leq\)"-sign is again to be interpreted elementwise.

**Example 2.18** Reconsider the Petri net from Example 2.16. Let's assume that the specification is still given by

\[x_2(k) + 3x_4(k) \leq 3, \quad k = 0, 1, \ldots\]
but that transition $t_4$ is now uncontrollable. Hence

$$A = \begin{bmatrix} A_{oc} & A_{ouc} \end{bmatrix} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. $$

Clearly, the specification is not ideally enforceable as (2.23) is violated. We therefore try to come up with a stricter and ideally enforceable specification using the procedure outlined above. For

$$R_1 = \begin{bmatrix} 0 & 0 & 3 & 0 \end{bmatrix}$$

and

$$R_2 = 1$$

the required conditions hold, and the “new” specification is given by

$$\Gamma = \begin{bmatrix} 0 & 1 & 3 & 3 \end{bmatrix},
\quad \bar{b} = 3.$$ 

Fig. 2.21 illustrates that the new specification is indeed stricter than the original one. The ideal controller for the new specification is given by

$$x_c^0 = \bar{b} - \Gamma x^0
= 3$$

and

$$A_c = -\Gamma A
= \begin{bmatrix} -1 & -3 & 1 & 0 \end{bmatrix}. $$
As \((A_c)_{14} = 0\), there is no arc from the controller place to the uncontrollable transition \(t_4\), indicating that the new specification is indeed ideally enforceable. The resulting closed-loop system is shown in Fig. 2.22.

Figure 2.22: Closed loop for Example 2.18
TIMED PETRI NETS

A Petri net \((N, x^0)\), as discussed in the previous chapter, only models the ordering of the firings of transitions, but not the actual firing time. If timing information is deemed important, we have to “attach” it to the “logical” DES model \((N, x^0)\). This can be done in two ways: we can associate time information with transitions or with places.

3.1 TIMED PETRI NETS WITH TRANSITION DELAYS

In this framework, the set of transitions, \(T\), is partitioned as

\[
T = T_W \cup T_D.
\]

A transition \(t_j \in T_W\) can fire without delay once the respective “logical” firing condition is satisfied, i.e., if (2.10) holds. A transition from \(T_D\) can only fire if both the “logical” firing condition is satisfied and a certain delay has occurred. The delay for the \(k\)-th firing of \(t_j \in T_D\) is denoted by \(v_{jk}\), and the sequence of delays by

\[
v_j := v_{j1} v_{j2} \ldots
\]

Definition 3.1 A timed Petri net with transition delays is a triple \((N, x^0, V)\), where

\[
(N, x^0) \ldots \text{ a Petri net}
\]

\[
T = T_W \cup T_D \ldots \text{ a partitioned set of transitions}
\]

\[
V = \{v_1, \ldots, v_{m_D}\} \ldots \text{ a set of sequences of time delays}
\]

\[
m_D = |T_D| \ldots \text{ number of delayed transitions}
\]

If the delays for all firings of a transition \(t_j\) are identical, the sequence \(v_j\) reduces to a constant.

To distinguish delayed and undelayed transitions in graphical representations of the Petri net, the former are depicted by boxes instead of bars (see Figure 3.1).
3 Timed Petri Nets

![Timed Petri Net Diagram]

Figure 3.1: Graphical representation of delayed (left) and undelayed (right) transitions

3.2 **Timed Event Graphs with Transition Delays**

Recall that event graphs represent a special class of Petri nets. They are characterised by the fact that each place has exactly one input transition and one output transition and that all arcs have weight 1. For timed event graphs, we can give an explicit equation relating subsequent firing instants of transitions. To see this, consider Figure 3.2 which shows part of a general timed event graph. Let’s introduce the following additional notation:

![Timed Event Graph Diagram]

Figure 3.2: Part of a general timed event graph.

- \( \tau_j(k) \) \ldots earliest possible time for the \( k \)-th firing of transition \( t_j \)
- \( \pi_i(k) \) \ldots earliest possible time for place \( p_i \) to receive its \( k \)-th token.

Then:

\[
\pi_i(k + x^0_i) = \tau_r(k), \quad t_r \in I(p_i), \quad k = 1, 2, \ldots \quad (3.1)
\]

\[
\tau_j(k) = \max_{p_i \in I(t_j)} (\pi_i(k)) + v_{ji}, \quad k = 1, 2, \ldots \quad (3.2)
\]

(3.1) says that, because of the initial marking \( x^0_i \), the place \( p_i \) will receive its \( (k + x^0_i) \)-th token when its input transition \( t_r \) fires for the \( k \)-th time. The earliest time instant for this to happen is \( \tau_r(k) \).


3.2 Timed Event Graphs with Transition Delays

(3.2) says that transition $t_j$ cannot fire the $k$-th time before all its input places have received their $k$-th token and the delay $v_{j_k}$ has passed.

We can now eliminate $\pi_i(k)$, $i = 1, \ldots, n$ from (3.1) and (3.2) to get the desired relation. This is illustrated in the following example.

Example 3.1 Consider the timed event graph shown in Fig. 3.3.

![Timed Event Graph with Transition Delays](image)

Figure 3.3: Example of a timed event graph with transition delays.

We get:

$$
\tau_1(k) = \max(\tau_1(k), \tau_3(k)) \quad (3.3)
$$

$$
\tau_2(k) = \pi_2(k) + v_{2k} \quad (3.4)
$$

$$
\pi_1(k+1) = \tau_1(k) \quad (3.5)
$$

$$
\pi_2(k+1) = \tau_1(k) \quad (3.6)
$$

$$
\pi_3(k) = \tau_2(k) \quad . \quad (3.7)
$$

We can now eliminate $\pi_1$, $\pi_2$, and $\pi_3$ from (3.3) to (3.7). We first insert (3.5) and (3.7) in (3.3) to give

$$
\tau_1(k+1) = \max(\tau_1(k), \tau_2(k+1)).
$$

Inserting (3.4) and subsequently (3.6) results in

$$
\tau_1(k+1) = \max(\tau_1(k), \tau_1(k) + v_{2k+1})
$$

$$
= \tau_1(k) + v_{2k+1}.
$$

Inserting (3.6) into (3.4) gives

$$
\tau_2(k+1) = \tau_1(k) + v_{2k+1}
$$

Note that the initial condition for the above difference equations is $\tau_1(1) = \tau_2(1) = v_{2i}$. ♦
3.3 Timed Petri Nets with Holding Times

Now, we consider a different way of associating time with a Petri net. We partition the set of places, $P$, as

$$P = P_W \cup P_D.$$ 

A token in a place $p_i \in P_W$ contributes without delay towards satisfying (2.10). In contrast, tokens in a place $p_i \in P_D$ have to be held for a certain time (“holding time”) before they contribute to enabling output transitions of $p_i$. We denote the holding time for the $k$-th token in place $p_i$ by $w_{i_k}$, and the sequence of holding times

$$w_i := w_{i_1} w_{i_2} \ldots$$

**Definition 3.2** A timed Petri net with holding times is a triple $(N, x^0, W)$, where

$$(N, x^0) \quad \ldots \quad \text{a Petri net}$$

$$P = P_W \cup P_D \quad \ldots \quad \text{a partitioned set of places}$$

$$W = \{w_1, \ldots, w_{n_D}\} \quad \ldots \quad \text{a set of sequences of holding times}$$

$$n_D = |P_D| \quad \ldots \quad \text{number of places with delays}$$

If the holding times for all tokens in a place $p_i$ are identical, the sequence $w_i$ reduces to a constant.

In graphical representations, places with and without holding times are distinguished as indicated in Figure 3.4.

![Figure 3.4: Graphical representation of places with holding times (left) and places without delays (right)](image)

3.4 Timed Event Graphs with Holding Times

For timed event graphs with transition delays, we could explicitly relate the times of subsequent firings of transitions. This is also
3.4 Timed Event Graphs with Holding Times

possible for timed event graphs with holding times. To see this, consider Figure 3.5 which shows a part of a general timed event graph with holding times.

We now have

\[ p_i(k + x_i^0) = \tau_r(k), \quad t_r \in I(p_i), \quad k = 1, 2, \ldots \] (3.8)
\[ \tau_j(k) = \max_{p_i \in I(t_j)} (\pi_i(k) + w_i), \quad k = 1, 2, \ldots \] (3.9)

(3.9) says that the earliest possible instant of the k-th firing for transition \( t_j \) is when all its input places have received their k-th token and the corresponding holding times \( w_i \) have passed.

(3.8) says that place \( p_i \) will receive its \((k + x_i^0)\)-th token when its input transition \( t_r \) fires for the k-th time.

As in Section 3.2 we can eliminate the \( \pi_i(k), i = 1, \ldots, n \), from (3.8) and (3.9) to provide the desired explicit relation between subsequent firing instants of transitions.

Remark 3.1 In timed event graphs, transition delays can always be “transformed” into holding times (but not necessarily the other way around). It is easy to see how this can be done: we just “shift” each transition delay \( v_j \) to all the input places of the corresponding transition \( t_j \). As each place has exactly one output transition, this will not cause any inconsistency.

Example 3.2 Consider the timed event graph with transition delays in Figure 3.3. Applying the procedure described above provides the timed event graph with holding times \( w_2 = v_2, i = 1, 2, \ldots \), shown in Figure 3.6. It is a simple exercise to determine the recursive equations for the earliest firing times of transitions, \( \tau_1(k), \tau_2(k), k = 1, 2, \ldots \), for this graph. Not surprisingly we get the same equations as in Example 3.1, indicating that the obtained timed event graph with holding times is indeed equivalent to the original timed event graph with transition delays. ∎
3 Timed Petri Nets

![Diagram of Timed Petri Nets]

Figure 3.6: Equivalent timed event graph with holding times.

3.5 THE MAX-PLUS ALGEBRA

From the discussion in Sections 3.2 and 3.4, it is clear that we can recursively compute the earliest possible firing times for transitions in timed event graphs. In the corresponding equations, two operations were needed: max and addition. This fact was the motivation for the development of a systems and control theory for a specific algebra, the so-called max-plus algebra, where these equations become linear. A good survey on this topic is [4] and the book [1]. We start with an introductory example, which is taken from [2].

3.5.1 Introductory example

Imagine a simple public transport system with three lines (see Fig: 3.7): an inner loop and two outer loops. There are two stations where passengers can change lines, and four rail tracks connecting the stations. Initially, we assume that the transport company operates one train on each track. A train needs 3 time units to travel on the inner loop from station 1 to station 2, 5 time units to travel on the inner loop from station 2 to station 1, 3 time units to travel on the outer loop from station 1 to station 2, and 3 time units to travel on the outer loop from station 2 to station 1.

![Diagram of Simple Train Example]

Figure 3.7: Simple train example (from [2]).

---

1 A pdf-version of this book is available for free on the web at http://cermics.enpc.fr/~cohen-g//SED/book-online.html
units for the track from station 2 to station 1, and 2 and 3 time units for the outer loops, respectively. We want to implement a user-friendly policy where trains wait for each other at the stations to allow passengers to change lines without delay. This can be easily represented in a timed event, or synchronization, graph with holding times (Figure 3.8). It is now straightforward to determine the recursive equations for the firing instants of transitions \( t_1 \) and \( t_2 \). These are the times when trains may leave the respective stations and can therefore be interpreted as the “time table” for our simple public transport system. We get

\[
\tau_1(k) = \max(\pi_1(k) + 2, \pi_4(k) + 5) \quad (3.10)
\]

\[
\tau_2(k) = \max(\pi_2(k) + 3, \pi_3(k) + 3) \quad (3.11)
\]

and

\[
\pi_1(k + x_1^0) = \pi_1(k + 1) = \tau_1(k) \quad (3.12)
\]

\[
\pi_2(k + x_2^0) = \pi_2(k + 1) = \tau_1(k) \quad (3.13)
\]

\[
\pi_3(k + x_3^0) = \pi_3(k + 1) = \tau_2(k) \quad (3.14)
\]

\[
\pi_4(k + x_4^0) = \pi_4(k + 1) = \tau_2(k) \quad (3.15)
\]

Inserting (3.12)–(3.15) into (3.10), (3.11) gives

\[
\tau_1(k + 1) = \max(\pi_1(k) + 2, \tau_2(k) + 5) \quad (3.16)
\]

\[
\tau_2(k + 1) = \max(\tau_1(k) + 3, \tau_2(k) + 3) \quad (3.17)
\]

for \( k = 1, 2, \ldots \). Let’s assume \( \tau_1(1) = \tau_2(1) = 0 \), i.e., trains leave both stations 1 and 2 at time 0 for the first time. Then, subsequent departure times are

\[
\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 5 \\ 3 \end{pmatrix}, \begin{pmatrix} 8 \\ 8 \end{pmatrix}, \begin{pmatrix} 13 \\ 11 \end{pmatrix}, \begin{pmatrix} 16 \\ 16 \end{pmatrix}, \ldots
\]

On the other hand, if the initial departure times are \( \tau_1(1) = 1 \) and \( \tau_2(1) = 0 \), we get the sequence

\[
\begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 5 \\ 4 \end{pmatrix}, \begin{pmatrix} 9 \\ 8 \end{pmatrix}, \begin{pmatrix} 13 \\ 12 \end{pmatrix}, \begin{pmatrix} 17 \\ 16 \end{pmatrix}, \ldots
\]
Hence, in the second case, trains leave every 4 time units from both stations (1-periodic behaviour), whereas in the first case the interval between subsequent departures changes between 3 and 5 time units (2-periodic behaviour). In both cases, the average departure interval is 4. This is of course not surprising, because a train needs 8 time units to complete the inner loop, and we operate two trains in this loop. Hence, it is obvious what to do if we want to realise shorter departure intervals: we add another train on the inner loop, initially, e.g., on the track connecting station 1 to station 2. This changes the initial marking of the timed event graph in Figure 3.8 to \( x^0 = (1, 2, 1, 1)' \). Equation 3.13 is now replaced by

\[
\pi_2(k + x_2^0) = \pi_2(k + 2) = \tau_1(k) \quad (3.18)
\]

and the resulting difference equations for the transition firing times are

\[
\begin{align*}
\tau_1(k + 1) &= \max(\tau_1(k) + 2, \tau_2(k) + 5) \quad (3.19) \\
\tau_2(k + 2) &= \max(\tau_1(k) + 3, \tau_2(k + 1) + 3) \quad (3.20)
\end{align*}
\]

for \( k = 1, 2, \ldots \). By introducing the new variable \( \tau_3 \), with \( \tau_3(k + 1) := \tau_1(k) \), we transform (3.19), (3.20) again into a system of first order difference equations:

\[
\begin{align*}
\tau_1(k + 1) &= \max(\tau_1(k) + 2, \tau_2(k) + 5) \quad (3.21) \\
\tau_2(k + 1) &= \max(\tau_3(k) + 3, \tau_2(k) + 3) \quad (3.22) \\
\tau_3(k + 1) &= \max(\tau_1(k)) \quad . \quad (3.23)
\end{align*}
\]

If we initialise this system with \( \tau_1(1) = \tau_2(1) = \tau_3(1) = 0 \), we get the following evolution:

\[
\begin{pmatrix}
0 \\
0 \\
0
\end{pmatrix},
\begin{pmatrix}
5 \\
3 \\
0
\end{pmatrix},
\begin{pmatrix}
8 \\
6 \\
5
\end{pmatrix},
\begin{pmatrix}
11 \\
9 \\
8
\end{pmatrix},
\begin{pmatrix}
14 \\
12 \\
11
\end{pmatrix}, \ldots
\]

We observe that after a short transient period, trains depart from both stations in intervals of three time units. Obviously, shorter intervals cannot be reached for this configuration, as now the right outer loop represents the “bottleneck”.

In this simple example, we have encountered a number of different phenomena: 1-periodic solutions (for \( \tau_1(1) = 1, \tau_2(1) = 0 \)), 2-periodic solutions (for \( \tau_1(1) = \tau_2(1) = 0 \)) and a transient phase (for the extended system). These phenomena (and more) can be conveniently analysed and explained within the formal framework of max-plus algebra.

\( \diamond \)
3.5 The Max-Plus Algebra

3.5.2 Max-Plus Basics

**Definition 3.3 (Max-Plus Algebra)** The max-plus algebra consists of the set \( R := \mathbb{R} \cup \{-\infty\} \) and two binary operations on \( R \):

\( \oplus \) is called the addition of max-plus algebra and is defined by
\[
 a \oplus b = \max(a, b) \quad \forall a, b \in R.
\]

\( \otimes \) is called multiplication of the max-plus algebra and is defined by
\[
 a \otimes b = a + b \quad \forall a, b \in R.
\]

The following properties are obvious:

- \( \oplus \) and \( \otimes \) are commutative, i.e.
  \[
  a \oplus b = b \oplus a \quad \forall a, b \in R \quad \text{and} \quad a \otimes b = b \otimes a \quad \forall a, b \in R.
  \]

- \( \oplus \) and \( \otimes \) are associative, i.e.
  \[
  (a \oplus b) \oplus c = a \oplus (b \oplus c) \quad \forall a, b, c \in R \quad \text{and} \quad (a \otimes b) \otimes c = a \otimes (b \otimes c) \quad \forall a, b, c \in R.
  \]

- \( \otimes \) is distributive over \( \oplus \), i.e.
  \[
  (a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c) \quad \forall a, b, c \in R.
  \]

- \( \varepsilon := -\infty \) is the neutral element w.r.t. \( \oplus \), i.e.
  \[
  a \oplus \varepsilon = a \quad \forall a \in R.
  \]

\( \varepsilon \) is also called the zero-element of max-plus algebra.

- \( e := 0 \) is the neutral element w.r.t. \( \otimes \), i.e.
  \[
  a \otimes e = a \quad \forall a \in R.
  \]

\( e \) is also called the one-element of max-plus algebra.

- \( \varepsilon \) is absorbing for \( \otimes \), i.e.
  \[
  a \otimes \varepsilon = \varepsilon \quad \forall a \in R.
  \]

- \( \oplus \) is idempotent, i.e.
  \[
  a \oplus a = a \quad \forall a \in R.
  \]
This makes the max-plus algebra an idempotent semi-field. Note that the idempotency property of $\oplus$ implies that there is no additive inverse, i.e., one cannot subtract. To see this, assume that there exists an inverse element, denoted $\bar{a}$, for $a$, i.e.,

$$a \oplus \bar{a} = \varepsilon.$$

Adding $a$ to both sides of the equation gives

$$a \oplus a \oplus \bar{a} = a \oplus \varepsilon.$$

Hence, the only element with an additive inverse is $\varepsilon$. It is straightforward to extend both $\oplus$ and $\otimes$ to matrices with elements in $\mathbb{R}$:

- matrix addition: let $A, B \in \mathbb{R}^{m \times n}$ with elements $a_{ij}, b_{ij}$. Then,

$$\begin{align*}
(A \oplus B)_{ij} &:= a_{ij} \oplus b_{ij} \\
&= \max(a_{ij}, b_{ij})
\end{align*}$$

- matrix multiplication: let $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times q}$. Then,

$$\begin{align*}
(A \otimes B)_{ij} &:= \bigoplus_{k=1}^{n} (a_{ik} \otimes b_{kj}) \\
&= \max_{k=1,\ldots,n} (a_{ik} + b_{kj})
\end{align*}$$

- multiplication with a scalar: let $A \in \mathbb{R}^{m \times n}, \alpha \in \mathbb{R}$. Then,

$$\begin{align*}
(\alpha \otimes A)_{ij} &:= \alpha \otimes a_{ij} \\
&= \alpha + a_{ij}
\end{align*}$$

- null and identity matrix:

$$N := \begin{bmatrix} \varepsilon & \cdots & \varepsilon \\ \vdots & \ddots & \vdots \\ \varepsilon & \cdots & \varepsilon \end{bmatrix}$$

is the null matrix and

$$E := \begin{bmatrix} \varepsilon & \varepsilon & \cdots & \varepsilon \\ \varepsilon & \varepsilon & \ddots & \vdots \\ \vdots & \ddots & \ddots & \varepsilon \\ \varepsilon & \cdots & \varepsilon & \varepsilon \end{bmatrix}$$

is the identity matrix.

As in standard algebra, we will often omit the multiplication symbol, i.e., $AB$ will mean $A \otimes B$. 

46
3.5 The Max-Plus Algebra

3.5.3 Max-plus algebra and precedence graphs

With each square matrix with elements in \( \mathbb{R} \) we can uniquely associate its precedence graph.

**Definition 3.4 (Precedence Graph)** Let \( A \in \mathbb{R}^{n \times n} \). Its precedence graph \( G(A) \) is a weighted directed graph with \( n \) nodes, labelled \( 1, \ldots, n \), with an arc from node \( j \) to node \( i \) if \( a_{ij} \neq \epsilon \); \( i, j = 1, \ldots, n \). If an arc from node \( j \) to node \( i \) exists, its weight is \( a_{ij} \).

**Example 3.3** Consider the \( 5 \times 5 \) matrix

\[
A = \begin{pmatrix}
\epsilon & 5 & \epsilon & 2 & \epsilon \\
\epsilon & \epsilon & 8 & \epsilon & 2 \\
\epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\
\epsilon & 3 & 7 & \epsilon & 4 \\
\epsilon & \epsilon & 4 & \epsilon & \epsilon
\end{pmatrix}
\]  
(3.24)

The precedence graph has 5 nodes, and the \( i \)-th row of \( A \) represents the arcs ending in node \( i \) (Figure 3.9).

![Figure 3.9: Precedence graph for (3.24)](image)

**Definition 3.5 (Path)** A path \( \rho \) in \( G(A) \) is a sequence of nodes \( i_1, \ldots, i_p, \ p > 1 \), with arcs from node \( i_j \) to node \( i_{j+1}, \ j = 1, \ldots, p - 1 \). The length of a path \( \rho = i_1, \ldots, i_p \), denoted by \( |\rho|_L \), is the number of its arcs. Its weight, denoted by \( |\rho|_W \), is the sum of the weights of its arcs, i.e.,

\[
|\rho|_L = p - 1 \\
|\rho|_W = \sum_{j=1}^{p-1} a_{i_{j+1}i_j}
\]

A path is called elementary, if all its nodes are distinct.

**Definition 3.6 (Circuit)** A path \( \rho = i_1, \ldots, i_p, \ p > 1 \), is called a circuit, if its initial and its final node coincide, i.e., if \( i_1 = i_p \). A circuit \( \rho = i_1, \ldots, i_p \) is called elementary, if the path \( \tilde{\rho} = i_1, \ldots, i_{p-1} \) is elementary.
Example 3.4 Consider the graph in Figure 3. Clearly, $\rho = 3, 5, 4, 1$ is a path with length 3 and weight 10. The graph does not contain any circuits.

Remark 3.2 The above definitions imply that, although a circuit is a path, an elementary circuit is of course not an elementary path.

For large graphs, it may be quite cumbersome to check “by inspection” whether circuits exist. Fortunately, this is straightforward in the max-plus framework. To see this, consider the product

$$A^2 := A \otimes A.$$  

By definition, $(A^2)_{ij} = \max_k (a_{ik} + a_{kj})$, i.e., the $(i, j)$-element of $A^2$ represents the maximal weight of all paths of length 2 from node $j$ to node $i$ in $G(A)$. More generally, $(A^k)_{ij}$ is the maximal weight of all paths of length $k$ from node $j$ to node $i$ in $G(A)$. Then it is easy to prove the following:

Theorem 3.1 $G(A)$ does not contain any circuits if and only if $A^k = N \forall k \geq n$.

Proof First assume that there are no circuits in $G(A)$. As $G(A)$ has $n$ nodes, this implies that there is no path of length $k \geq n$, hence $A^k = N \forall k \geq n$. Now assume that $A^k = N \forall k \geq n$, i.e., there exists no path in $G(A)$ with length $k \geq n$. As a circuit can always be extended to an arbitrarily long path, this implies the absence of circuits.

Example 3.5 Consider the $5 \times 5$-matrix $A$ from Example 3.3 and its associated precedence graph $G(A)$. Matrix multiplication provides

$$A^2 = \begin{pmatrix}
\epsilon & 5 & 13 & \epsilon & 7 \\
\epsilon & \epsilon & 6 & \epsilon & \epsilon \\
\epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\
\epsilon & \epsilon & 11 & \epsilon & 5 \\
\epsilon & \epsilon & \epsilon & \epsilon & \epsilon
\end{pmatrix}$$

$$A^3 = \begin{pmatrix}
\epsilon & \epsilon & 13 & \epsilon & 7 \\
\epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\
\epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\
\epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\
\epsilon & \epsilon & 9 & \epsilon & \epsilon
\end{pmatrix}$$
The Max-Plus Algebra

\[
A^4 = \begin{pmatrix}
\varepsilon & \varepsilon & 11 & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon & \varepsilon & \varepsilon \\
\end{pmatrix}
\]

\[
A^5 = N.
\]

This implies that there are only three pairs of nodes between which paths of length 3 exist. For example, such paths exist from node 3 to node 1, and the one with maximal length (13) is \( \rho = 3, 2, 4, 1 \). As expected, there is no path of length 5 or greater, hence no circuits exist in \( \mathcal{G}(A) \).

3.5.4 Linear implicit equations in max-plus

In the following we will often encounter equations of the form

\[
x = Ax \oplus b, \quad (3.25)
\]

where \( A \in \mathbb{R}^{n \times n} \) and \( b \in \mathbb{R}^n \) are given and a solution for \( x \) is sought. We will distinguish three cases:

1. \( \mathcal{G}(A) \) does not contain any circuits. Repeatedly inserting \( (3.25) \) into itself provides

\[
x = A(Ax \oplus b) \oplus b = A^2x \oplus Ab \oplus b
\]

\[
= A^2(Ax \oplus b) \oplus Ab \oplus b = A^3x \oplus A^2b \oplus Ab \oplus b
\]

\[\vdots\]

\[
x = A^nx \oplus A^{n-1}b \oplus \ldots \oplus Ab \oplus b.
\]

As \( A^n = N \), we get the unique solution

\[
x = \left( E \oplus A \oplus \ldots \oplus A^{n-1} \right) b. \quad (3.26)
\]

2. All circuits in \( \mathcal{G}(A) \) have negative weight. As before, we repeatedly insert \( (3.25) \) into itself. Unlike in the previous case, we do not have \( A^n = N \), hence we keep inserting:

\[
x = \left( \lim_{k \to \infty} A^k \right) x \oplus \left( E \oplus A \oplus A^2 \oplus \ldots \right) b
\]

\[:= A^* \]

Note that \( \left( \lim_{k \to \infty} A^k \right)_{ij} \) represents the maximum weight of infinite-length paths from node \( j \) to node \( i \) in \( \mathcal{G}(A) \). Clearly, such paths, if they exist, have to contain an infinite number
of elementary circuits. As all these circuits have negative weight, we get

$$\lim_{k \to \infty} A^k = N.$$  \hfill (3.27)

With a similar argument, it can be shown that in this case

$$A^* = E \oplus A \oplus \ldots \oplus A^{n-1}. \hfill (3.28)$$

To see this, assume that \((A^k)_{ij} \neq \varepsilon\) for some \(i, j\) and some \(k \geq n\), i.e., there exists a path \(\rho\) of length \(k \geq n\) from node \(j\) to node \(i\). Clearly, this path must contain at least one circuit and can therefore be decomposed into an elementary path \(\tilde{\rho}\) of length \(l < n\) from \(j\) to \(i\) and one or more circuits. As all circuits have negative weights, we have for all \(k \geq n\)

$$(A^k)_{ij} = |\rho|_W < |\tilde{\rho}|_W = (A^l)_{ij} \hbox{ for some } l < n.$$ \hfill (3.28)

follows immediately. Hence, \((3.26)\) is also the unique solution if all circuits in \(G(A)\) have negative weight.

3. All circuits in \(G(A)\) have non-positive weights. We repeat the argument from the previous case and decompose any path \(\rho\) of length \(k \geq n\) into an elementary path \(\tilde{\rho}\) and at least one circuit. We get that for all \(k \geq n\)

$$(A^k)_{ij} = |\rho|_W \leq |\tilde{\rho}|_W = (A^l)_{ij} \hbox{ for some } l < n$$

and therefore, in this case also,

$$A^* = E \oplus A \oplus \ldots \oplus A^{n-1}$$

Furthermore, it can be easily shown that \(x = A^*b\) represents a (not necessarily unique) solution to \((3.25)\). To see this we just insert \(x = A^*b\) into \((3.25)\) to get

$$A^*b = A(A^*b) \oplus b = (E \oplus AA^*)b = (E \oplus A \oplus A^2 \oplus \ldots)b = A^*b$$

In summary, if the graph \(G(A)\) does not contain any circuits with positive weights, \((3.26)\) represents a solution for \((3.25)\). If all circuits have negative weights or if no circuits exist, this is the unique solution.
3.5 The Max-Plus Algebra

3.5.5 State equations in max-plus

We now discuss how timed event graphs with some autonomous transitions can be modelled by state equations in the max-plus algebra. We will do this for an example which is taken from [1].

Example 3.6 Consider the timed event graph with holding times in Figure 3.10.\(\text{t}_1\) and \(\text{t}_2\) are autonomous transitions, i.e., their firing does not depend on the marking of the Petri net. The firing of these transitions can therefore be interpreted as an input, and the firing times are denoted by \(u_1(k), u_2(k), k = 1, 2, \ldots\), respectively. The firing times of transition \(\text{t}_6\) are considered to be an output and therefore denoted \(y(k)\). Finally, we denote the \(k\)-th firing times of transitions \(\text{t}_3, \text{t}_4\) and \(\text{t}_5\) by \(x_1(k), x_2(k)\) and \(x_3(k)\), respectively.

![Timed event graph with holding times and autonomous transitions](image-url)
As discussed in Section 3.4, we can explicitly relate the firing times of the transitions:

\[
\begin{align*}
x_1(k+1) &= \max(u_1(k+1) + 1, x_2(k) + 4) \\
x_2(k+1) &= \max(u_2(k) + 5, x_1(k+1) + 3) \\
x_3(k+1) &= \max(x_3(k-1) + 2, x_2(k+1) + 4, x_1(k+1) + 3) \\
y(k+1) &= \max(x_2(k), x_3(k+1) + 2)
\end{align*}
\]

In vector notation, i.e.,

\[
\begin{align*}
x(k) &:= (x_1(k), x_2(k), x_3(k))' \\
u(k) &:= (u_1(k), u_2(k))'
\end{align*}
\]

this translates into the following max-plus equations:

\[
\begin{align*}
x(k+1) &= \begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \\ 3 & \varepsilon & \varepsilon \\ 3 & 4 & \varepsilon \end{pmatrix} x(k+1) \oplus \begin{pmatrix} \varepsilon & 4 & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon \end{pmatrix} x(k) \\
&\quad \oplus \begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & 2 \end{pmatrix} x(k-1) \oplus \begin{pmatrix} 1 & \varepsilon \\ \varepsilon & \varepsilon \\ \varepsilon & \varepsilon \end{pmatrix} u(k+1) \\
&\quad \oplus \begin{pmatrix} \varepsilon & \varepsilon & 5 \\ \varepsilon & \varepsilon \end{pmatrix} u(k) \\
&\quad := A_0 \\
&\quad := A_1 \\
&\quad := A_2 \\
y(k) &= \begin{pmatrix} \varepsilon & \varepsilon & 2 \end{pmatrix} x(k) \oplus \begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \end{pmatrix} x(k-1) \quad := C_0 \quad := C_1
\end{align*}
\]

In a first step, we convert (3.29) into explicit form. Clearly, \( G(A_0) \) does not contain any circuits (see Fig. 3.11), therefore \( A_0^* = E \oplus A_0 \oplus A_0^2 \) and

\[
\begin{align*}
x(k+1) &= A_0^* (A_1 x(k) \oplus A_2 x(k-1) \oplus B_0 u(k+1) \oplus B_1 u(k)) \\
&= \begin{pmatrix} \varepsilon & 4 & \varepsilon \\ \varepsilon & 7 & \varepsilon \\ \varepsilon & 11 & \varepsilon \end{pmatrix} x(k) \oplus \begin{pmatrix} \varepsilon & \varepsilon & \varepsilon \\ \varepsilon & \varepsilon & 2 \\ \varepsilon & \varepsilon & \varepsilon \end{pmatrix} x(k-1) \\
&\quad \oplus \begin{pmatrix} 1 & \varepsilon \\ 4 & \varepsilon \\ 8 & \varepsilon \end{pmatrix} u(k+1) \oplus \begin{pmatrix} \varepsilon & \varepsilon & 5 \\ \varepsilon & \varepsilon & 9 \end{pmatrix} u(k) \\
&\quad := \tilde{A}_0 \\
&\quad := \tilde{A}_1 \\
&\quad := \tilde{A}_2 \\
&\quad := \tilde{B}_0 \\
&\quad := \tilde{B}_1
\end{align*}
\]
3.5 The Max-Plus Algebra

![Figure 3.11: $G(A_0)$ for Example 3.6](image)

is the desired explicit form. In a second step, we define an extended vector $\bar{x}(k) := (x'(k), x'(k - 1), u'(k))^T$ to get

$$
\bar{x}(k + 1) = \begin{pmatrix}
A_1 & A_2 & B_1 \\
E & N & N \\
N & N & N \\
\end{pmatrix}
\bar{x}(k) \oplus
\begin{pmatrix}
B_0 \\
N \\
E \\
\end{pmatrix}
\quad u(k + 1)
$$

$$y(k) = \begin{pmatrix}
C_0 & C_1 & N \\
\end{pmatrix}
\bar{x}(k).
$$

3.5.6 The max-plus eigenproblem

Recall the introductory example in Section 3.5.1. Depending on the vector of initial firing times, we observed a number of different phenomena: 1- and 2-periodic behaviour with and without an initial transient phase. For many application scenarios as, e.g., the one envisaged in the example, a 1-periodic solution is desirable. It is therefore natural to ask, which initial firing vectors will indeed generate 1-periodic solutions and what the duration for one period is.

Consider a timed event graph without autonomous transitions and assume that we have already converted the equations describing the firing times into a system of explicit first order difference equations (see Section 3.5.5), i.e.,

$$
x(k + 1) = Ax(k), \quad k = 1, 2, \ldots
$$

(3.31)

As $x(k)$ represents the (extended) vector of firing times, the requirement for a 1-periodic solution means in conventional algebra that

$$
x_i(k + 1) = \lambda + x_i(k), \quad k = 1, 2, \ldots
$$

$$i = 1, 2, \ldots, n.
$$
Timed Petri Nets

In the max-plus context this reads as

\[ x_i(k + 1) = \lambda \otimes x_i(k), \quad k = 1, 2, \ldots, \]

or, equivalently,

\[ x(k + 1) = \lambda \otimes x(k), \quad k = 1, 2, \ldots. \quad (3.32) \]

Let us now consider the eigenproblem in the max-plus algebra. If, for a given \( A \in \mathbb{R}^{n \times n} \), there exist \( \xi \in \mathbb{R}^n \) and \( \lambda \in \mathbb{R} \) such that

\[ A\xi = \lambda \xi, \quad (3.33) \]

we call \( \lambda \) eigenvalue and \( \xi \) eigenvector of the matrix \( A \). If we choose the vector of initial firing times, \( x(1) \), as an eigenvector, we get

\[ x(2) = Ax(1) = \lambda x(1) \]

and therefore

\[ x(k) = \lambda^{k-1} x(1), \quad k = 1, 2, \ldots. \]

This is the desired 1-periodic behaviour and the period length is the eigenvalue \( \lambda \).

To solve the max-plus eigenproblem, we need the notions of matrix (ir)reducibility and strong connectedness of graphs.

**Definition 3.7 (Ir)reducibility** The matrix \( A \in \mathbb{R}^{n \times n} \) is called reducible, if there exists a permutation matrix \( P \) such that

\[ \tilde{A} = PAP' \]

is upper block-triangular. Otherwise, \( A \) is called irreducible.

**Definition 3.8 (Strongly connected graph)** A directed graph is strongly connected, if there exists a path from any node \( i \) to any other node \( j \) in the graph.

**Remark 3.3** Definition 3.7 can be rephrased to say that the matrix \( A \) is reducible if it can be transformed to upper block-triangular form by simultaneously permuting rows and columns. Hence, \( A \) is reducible if and only if the index set \( I = \{1, \ldots, n\} \) can be partitioned as

\[ I = \{i_1, \ldots, i_k\} \cup \{i_{k+1}, \ldots, i_n\} \]

Recall that a permutation matrix is obtained by permuting the rows of the \( n \times n \)-identity matrix. In the max-plus context, this is of course the matrix \( E \) (Section 3.5.2).
such that

\[ a_{ij} = \varepsilon \quad \forall i \in I_1, j \in I_2. \]

This is equivalent to the fact that in the precedence graph \( G(A) \) there is no arc from any node \( j \in I_2 \) to any node \( i \in I_1 \). We therefore have the following result.

**Theorem 3.2** The matrix \( A \in \mathbb{R}^{n \times n} \) is irreducible if and only if its precedence graph \( G(A) \) is strongly connected.

**Example 3.7** Consider the matrix

\[
A = \begin{pmatrix}
1 & 2 & 3 \\
\varepsilon & 4 & \varepsilon \\
5 & 6 & 7
\end{pmatrix}.
\]

For

\[
P = \begin{pmatrix}
\varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon \\
\varepsilon & \varepsilon & \varepsilon
\end{pmatrix}
\]

we get

\[
\tilde{A} = PAP' = \begin{pmatrix}
1 & 3 & 2 \\
5 & 7 & 6 \\
\varepsilon & \varepsilon & 4
\end{pmatrix},
\]

which is clearly in upper block-triangular form. \( A \) is therefore reducible, and its precedence graph \( G(A) \) is not strongly connected. Indeed, there is no path from either node 1 or node 3 to node 2 (Figure 3.12).  

![Figure 3.12: Precedence graph for Example 3.7](image-url)
Theorem 3.3 If $A \in R^{n \times n}$ is irreducible, there exists precisely one eigenvalue. It is given by

$$\lambda = \bigoplus_{j=1}^{n} \left( \text{tr} \left( A^j \right) \right)^{1/j},$$

where "trace" and the $j$-th root are defined as in conventional algebra, i.e., for any $B \in R^{n \times n}$,

$$\text{tr}(B) = \bigoplus_{i=1}^{n} b_{ii}$$

and for any $\alpha \in R$,

$$\left( \alpha^{1/j} \right)^j = \alpha.$$

Proof See, e.g. [11].

Remark 3.4 [3.34] can also be interpreted in terms of the precedence graph $G(A)$: to do this, recall that $(A^j)_{ii}$ is the maximal weight of all circuits of length $j$ starting and ending in node $i$ of $G(A)$. Then,

$$\text{tr} \left( A^j \right) = \bigoplus_{i=1}^{n} (A^j)_{ii}$$

represents the maximum weight of all circuits of length $j$ in $G(A)$. Moreover, taking the $j$-th root in max-plus algebra corresponds to dividing by $j$ in conventional algebra, therefore

$$\left( \text{tr} \left( A^j \right) \right)^{1/j}$$

is the maximum mean weight (i.e. weight divided by length) of all circuits of length $j$. Finally, recall that the maximum length of any elementary circuit in $G(A)$ is $n$, and that the mean weight of any circuit can never be greater than the maximal mean weight of all elementary circuits. Therefore, [3.34] represents the maximal mean weight of all circuits in $G(A)$, or the maximal cycle mean, for short:

$$\bigoplus_{j=1}^{n} \left( \text{tr} \left( A^j \right) \right)^{1/j} = \max_{\rho \in S} \frac{|\rho|_W}{|\rho|_L},$$

where $S$ is the set of all circuits in $G(A)$.

Whereas an irreducible matrix $A \in R^{n \times n}$ has a unique eigenvalue $\lambda$, it may possess several distinct eigenvectors. In the following, we provide a scheme to compute them:
**Step 1** Scale the matrix $A$ by multiplying it with the inverse of its eigenvalue $\lambda$, i.e.,

$$Q := \text{inv}_\otimes(\lambda) \otimes A.$$  

Hence, in conventional algebra, we get $Q$ by subtracting $\lambda$ from every element of $A$. This implies that $\mathcal{G}(A)$ and $\mathcal{G}(Q)$ are identical up to the weights of their arcs. In particular, $\rho$ is a path (circuit) in $\mathcal{G}(A)$ if and only if it is a path (circuit) in $\mathcal{G}(Q)$. Let’s denote the weight of $\rho$ in $\mathcal{G}(A)$ and in $\mathcal{G}(Q)$ by $|\rho|_{W,A}$ and $|\rho|_{W,Q}$, respectively. Then, for any circuit $\rho$,

$$|\rho|_{W,Q} = |\rho|_{W,A} - |\rho|_L \cdot \lambda$$

$$\leq 0$$

as $\lambda$ is the maximum mean weight of all circuits in $\mathcal{G}(A)$. Hence, by construction, all circuits in $\mathcal{G}(Q)$ have nonpositive weight.

**Step 2** As shown in Section 3.5.4, (3.36) implies that

$$Q^* = E \oplus Q \oplus Q^2 \oplus \ldots = E \oplus Q \oplus \ldots \oplus Q^{n-1}$$

**Step 3** The matrix

$$Q^+ := Q \otimes Q^* \quad (3.37)$$

contains at least one diagonal element $q_{ii}^+ = e$. To see this, choose an elementary circuit $\tilde{\rho}$ in $\mathcal{G}(A)$ with maximal mean weight. Then (3.35) implies that the weight of $\tilde{\rho}$ in $\mathcal{G}(Q)$ is 0, i.e., $e$. Now choose any node $i$ in $\tilde{\rho}$. As the maximum length of any elementary circuit in $Q$ is $n$, $q_{ii}^+$ represents the maximal weight of all elementary circuits in $\mathcal{G}(Q)$ starting and ending in node $i$. Therefore, $q_{ii}^+ = e$.

**Step 4** If $q_{ii}^+ = e$, the corresponding column vector of $Q^+$, i.e., $q_i^+$, is an eigenvector of $A$. To see this, observe that

$$Q^* = E \oplus Q^+,$$

hence, the $j$-th entry of $q_i^+$ is

$$q_{ji}^* = \begin{cases} 
  e \oplus q_{ji}^+ & \text{for } j \neq i \\
  e \oplus q_{ji}^+ & \text{for } j = i 
\end{cases} = q_{ji}^+ \quad j = 1, \ldots, n.$$
as \( q_i^+ \) is assumed to be \( e \). Therefore, \( q_i^* = q_i^+ \). Furthermore, because of (3.37), we have

\[
q_i^+ = Q \otimes q_i^*
q_i^+ = Q \otimes q_i^+
q_i^+ = \text{inv} \otimes \lambda \otimes A \otimes q_i^+
\]

or, equivalently,

\[
\lambda \otimes q_i^+ = A \otimes q_i^+.
\]

**Example 3.8** Consider the matrix

\[
A = \begin{pmatrix}
\varepsilon & 5 & \varepsilon \\
3 & \varepsilon & 1 \\
\varepsilon & 1 & 4
\end{pmatrix}
\]

(3.38)

As the corresponding precedence graph \( G(A) \) is strongly connected (see Figure 3.13), \( A \) is irreducible. Therefore,

![Figure 3.13: Precedence graph for Example 3.8](image)

\[
\lambda = \bigoplus_{j=1}^{3} \text{tr} \left( A^j \right)
\]

\[
= 4
\]

is the unique eigenvalue of \( A \). To compute the eigenvectors, we follow the procedure outlined on the previous pages:

\[
Q = \text{inv} \otimes (\lambda) \otimes A
\]

\[
= \begin{pmatrix}
\varepsilon & 1 & \varepsilon \\
-1 & \varepsilon & -3 \\
\varepsilon & -3 & \varepsilon
\end{pmatrix}
\]

\[
Q^* = E \oplus Q \oplus Q^2
\]

\[
= \begin{pmatrix}
e & 1 & -2 \\
-1 & e & -3 \\
-4 & -3 & e
\end{pmatrix}
\]

\[
Q^+ = Q \otimes Q^*
\]

\[
= \begin{pmatrix}
e & 1 & -2 \\
-1 & e & -3 \\
-4 & -3 & e
\end{pmatrix}
\]
As all three diagonal elements of $Q^+$ are identical to $e$, all three columns are eigenvectors, i.e.

\[
\begin{bmatrix}
e \\
-1 \\
-4
\end{bmatrix}, \quad \begin{bmatrix}
1 \\
e \\
-3
\end{bmatrix}, \quad \begin{bmatrix}
-2 \\
-3 \\
e
\end{bmatrix}.
\]

Apparently,

\[
\xi_2 = 1 \otimes \xi_1,
\]
i.e. the eigenvectors $\xi_2$ and $\xi_1$ are linearly dependent, while $\xi_3$ and $\xi_1$ are not.

### 3.5.7 Linear independence of eigenvectors

Before we can clarify the phenomena of linear (in)dependence of eigenvectors, we need additional terminology from graph theory.

**Definition 3.9 (Critical circuit, critical graph)** A circuit $\rho$ in a weighted directed graph $G$ is called critical, if it has maximal mean weight of all circuits in $G$. The critical graph $G_c$ consists of all nodes and all arcs of all critical circuits in $G$.

**Definition 3.10 (Maximal strongly connected subgraph)** Let $G$ be a weighted directed graph with $I$ as set of nodes and $E$ as set of arcs. A graph $G'$ with node set $I'$ and arc set $E'$ is a (proper) subgraph of $G$, if $I' \subseteq I$ ($I' \subset I$) and if $E' = \{(i, j) | (i, j) \in E, i, j \in I'\}$. A subgraph $G'$ of $G$ is a maximal strongly connected (m.s.c.) subgraph, if it is strongly connected, and if it is not a proper subgraph of another strongly connected subgraph of $G$.

**Example 3.9** Consider the matrix

\[
A = \begin{pmatrix}
4 & 5 & e \\
3 & e & 1 \\
e & 1 & 4
\end{pmatrix}.
\]

Its precedence graph $G(A)$ is shown in Figure 3.14. The maximal mean weight of circuits is 4, hence the critical graph $G_c(A)$ consists of all circuits of mean weight 4 (Figure 3.15). Clearly, $G_c(A)$ has two m.s.c. subgraphs, $G_{c1}(A)$ and $G_{c2}(A)$.

We can now explain the phenomenon of linearly independent eigenvectors. Assume that $A \in R^{n \times n}$ is irreducible and therefore possesses precisely one eigenvalue $\lambda$. Using the procedure described in Section 3.5.6, we get a set of $m \leq n$ eigenvectors. More precisely, column $q_i^+$ of matrix $Q^+ = Q \oplus \ldots \oplus Q^n$ is an eigenvector of $A$, if its $i$-th entry is $e$. 

59
Theorem 3.4 Let $A \in \mathbb{R}^{n \times n}$ be irreducible and let the critical graph $G_c(A)$ consist of $N$ m.s.c. subgraphs $G_{c_j}(A)$ with node sets $I_j$, $j = 1, \ldots, N$. Then the following holds:

(i) If $i \in I := \bigcup_{j=1}^{N} I_j$, then $q_i^+$ is an eigenvector of $A$.

(ii) If $i_1, i_2 \in I_j$, then $q_{i_1}^+$ and $q_{i_2}^+$ are linearly dependent eigenvectors, i.e. $\exists \alpha \in \mathbb{R}$ s.t. $q_{i_1}^+ = \alpha \otimes q_{i_2}^+$.

(iii) If $i \in I_p$, then $q_i^+ \neq \bigoplus_{j \in I \setminus I_p} \alpha_j \otimes q_j^+$ for any set of $\alpha_j \in \mathbb{R}$.

Proof See, e.g., [1].

Example 3.10 Let’s reconsider the Example 3.8 where we determined three eigenvectors for (3.38). The critical graph for (3.38) is shown in Figure 3.16. It contains two m.s.c. subgraphs with

Figure 3.16: Critical graph $G_c(A)$ for (3.38).
node sets $I_1 = \{1, 2\}$ and $I_2 = \{3\}$. Hence,

$$
\xi_1 = q_1^+ = \begin{bmatrix} e \\ -1 \\ -4 \end{bmatrix} \quad \text{and} \quad \xi_2 = q_2^+ = \begin{bmatrix} 1 \\ e \\ -3 \end{bmatrix}
$$

are linearly dependent eigenvectors, whereas

$$
\xi_3 = q_3^+ = \begin{bmatrix} -2 \\ -3 \\ e \end{bmatrix}
$$

cannot be written as a linear combination of $q_1^+$ and $q_2^+$. ♦

### 3.5.8 Cyclicity

We have seen in the previous sections that the vectors of firing times in a timed event graph form a regular (1-periodic) behaviour, if the initial firing vector is an eigenvector of the matrix $A$. We also know from the motivating example (Section 3.5.1) that a transient phase and/or $k$-periodic ($k > 1$) behaviour may occur if the vector of initial firing times is not an eigenvector of $A$. To explain this, we need to introduce the notion of cyclicity of matrices in $\mathbb{R}^{n \times n}$.

**Definition 3.11 (Cyclicity)** Let $A \in \mathbb{R}^{n \times n}$ and let $\lambda$ be the maximal mean weight of all circuits in $G(A)$. If there exist positive integers $M, d$ such that

$$
A^{m+d} = \lambda^d \otimes A^m \quad \forall m \in \mathbb{N}, \ m \geq M \quad (3.39)
$$

the matrix $A$ is called cyclic. The smallest $d$ for which (3.39) holds is called the cyclicity of $A$.

**Remark 3.5** If $x(k+1) = Ax(k)$, with $x(k)$ the vector of the $k$-th firing instants, and if $A$ has cyclicity $d$ we will eventually observe $d$-periodic behaviour, irrespective of $x(1)$.

**Theorem 3.5** Each irreducible matrix $A$ is cyclic. If its critical graph $G_c(A)$ consists of $N$ m.s.c. subgraphs $G_{cj}(A)$, the cyclicity of $A$ is given by

$$
\text{cyc}(A) = \text{lcm}_{j=1,\ldots,N} \left[ \gcd_{\rho \in S(G_{cj}(A))} (|\rho|) \right] \quad (3.40)
$$

where $S(G_{cj}(A))$ is the set of all elementary circuits of $G_{cj}(A)$, gcd means "greatest common divisor" and lcm is "least common multiple".
Proof See, e.g., [1].

Example 3.11 Consider the matrix

\[
A = \begin{pmatrix}
\varepsilon & 5 & \varepsilon \\
3 & \varepsilon & 6 \\
\varepsilon & 2 & 4
\end{pmatrix}
\]  

(3.41)

with precedence graph \( G(A) \) shown in Figure 3.17. Clearly, the maximal mean circuit weight is 4, and all circuits are critical. Hence, \( G(A) = G_c(A) \). Obviously, \( G_c(A) \) is strongly connected, i.e., there is only one m.s.c. subgraph \( G_{11}(A) \), which is \( G_c(A) \) itself. We can then deduce from (3.40) that

\[ \text{cyc}(A) = 1. \]

Indeed, if we initialise the recursion \( x(k+1) = Ax(k) \) with a non-eigenvector of \( A \), e.g., \( x(1) = (0 \ 1 \ 2)' \), we get the following sequence of firing vectors:

\[
\begin{pmatrix}
0 \\
1 \\
2
\end{pmatrix} \rightarrow \begin{pmatrix}
6 \\
8 \\
6
\end{pmatrix} \rightarrow \begin{pmatrix}
13 \\
12 \\
10
\end{pmatrix} \rightarrow \begin{pmatrix}
17 \\
16 \\
14
\end{pmatrix} \rightarrow \begin{pmatrix}
21 \\
20 \\
18
\end{pmatrix}.
\]

Clearly, after a short transient phase, we get a 1-periodic behaviour where the period length is the maximal mean weight of all circuits in \( G(A) \), i.e., the eigenvalue of \( A \).

\[ \diamond \]

Example 3.12 Let’s reconsider our simple public transport system from Section 3.5.1. Figure 3.18 shows the precedence graph of the matrix

\[
A = \begin{pmatrix}
2 & 5 \\
3 & 3
\end{pmatrix}.
\]

Clearly, the maximal mean circuit weight is 4, therefore the critical graph \( G_c(A) \) consists of only one elementary circuit (Figure 3.19). Obviously, \( G_c(A) \) is strongly connected and therefore the only m.s.c. subgraph. Hence,

\[ \text{cyc}(A) = 2. \]

This explains the 2-periodic behaviour that we observed in Section 3.5.1.
Figure 3.18: Precedence graph for Example 3.12

Figure 3.19: Critical graph $G_c(A)$ for Example 3.12
3 Timed Petri Nets
The control of untimed (logical) DES has been an active area of research since the mid 1980s. It was shaped to a large extent by P.J. Ramadge and W.M. Wonham’s seminal work, e.g. [7] and [8]. Since then, numerous researchers have contributed to this area, which has come to be known as “Supervisory Control Theory (SCT)”. Standard references are [5], [3] and [9].

In this chapter, we will summarise the very basics of SCT. Briefly, the plant to be controlled is modelled as an untimed DES, and the controller design philosophy is language-based. This means that one is primarily interested in the set of event strings (“language”) that the plant can generate. It is then the aim of control to suitably restrict this set such that strings of events that are deemed to be undesirable cannot occur. At the same time, one wants to “keep” as many other strings of events as possible. In other words, the controller should only act if things (threaten to) go wrong. Although the philosophy of SCT is language-based, we have to keep in mind that control also needs to be realised. Hence, we will have to discuss finite state machines, or finite automata, as generators of languages.

### 4.1 SCT BASICS

Let’s assume that there is a finite set of discrete events

\[ \Sigma = \{ \sigma_1, \ldots, \sigma_N \} . \]  (4.1)

The events \( \sigma_i, i = 1, \ldots, N \), are also called symbols, and \( \Sigma \) is called an alphabet. Furthermore, denote the set of all finite strings of elements of \( \Sigma \), including \( \epsilon \) (the string of length 0), by \( \Sigma^* \), i.e.

\[ \Sigma^* = \{ \epsilon, \sigma_1, \ldots, \sigma_N, \sigma_1 \sigma_2, \sigma_1 \sigma_3, \ldots \} . \]  (4.2)

\( \Sigma^* \) is called the Kleene-closure of \( \Sigma \). Strings can be concatenated, i.e., if \( s, t \in \Sigma^* \), \( st \in \Sigma^* \) represents a string \( s \) followed by a string \( t \). Clearly, \( \epsilon \) is the neutral element of concatenation, i.e.,

\[ se = es = s \quad \forall s \in \Sigma^* . \]  (4.3)
Finally, a subset \( \mathcal{L} \subseteq \Sigma^* \) is called a language over the alphabet \( \Sigma \), and an element \( s \in \mathcal{L} \) is a word.

We can now define the concept of prefix and prefix-closure:

**Definition 4.1 (Prefix, prefix-closure)**: \( s' \in \Sigma^* \) is a prefix of a word \( s \in \mathcal{L} \), if there exists a string \( t \in \Sigma^* \) such that \( s't = s \). The set of all prefixes of all words in \( \mathcal{L} \) is called prefix-closure of \( \mathcal{L} \):

\[
\overline{\mathcal{L}} := \{ s' \in \Sigma^* \mid \exists t \in \Sigma^* \text{ such that } s't \in \mathcal{L} \} .
\]

By definition, every prefix can be extended into a word by appending suitable symbols from the alphabet. Note that every word \( s \in \mathcal{L} \) is a prefix of itself, as \( se = s \), but, in general, a prefix of a word is not a word. Therefore,

\[
\mathcal{L} \subseteq \overline{\mathcal{L}} .
\]

If \( \mathcal{L} = \overline{\mathcal{L}} \), the language \( \mathcal{L} \) is called closed. Hence, in a closed language every prefix of a word is a word.

### 4.2 Plant Model

A plant model has to provide the following information:

**Possible Future System Evolution**: In the context of untimed DES, this is the language \( \mathcal{L} \). Of course, a meaningful model will never allow all possible strings of events, and therefore \( \mathcal{L} \) will in practice always be a proper subset of \( \Sigma^* \).

**Control Mechanism**: In the context of SCT, the mechanism that a controller can use to affect the plant evolution is modelled by partitioning the event set \( \Sigma \) into a set of events that can be disabled by a controller, \( \Sigma_c \), and a set of events which cannot be directly prohibited, \( \Sigma_{uc} \):\[
\Sigma = \Sigma_c \cup \Sigma_{uc} ; \quad \Sigma_c \cap \Sigma_{uc} = \emptyset .
\]

\( \Sigma_c \) is often called the set of controllable events, whereas events in \( \Sigma_{uc} \) are called uncontrollable.

**Terminal Conditions**: As in “conventional” continuous control, it has become customary to include terminal conditions for the system evolution in the plant model. Of course, we could also interpret terminal conditions as specifications that a controller has to enforce. In the SCT context, such terminal conditions are modelled by a so-called marked language \( \mathcal{L}_m \subseteq \mathcal{L} \), which contains all strings of events that meet these conditions. These strings are called marked strings. In practice, one thinks of such strings as tasks that have successfully terminated.
In summary, the plant model is completely defined by

\[ P = (\Sigma = \Sigma_c \cup \Sigma_{uc}, L \subseteq \Sigma^*, L_m \subseteq L) \]

In the following, we will always assume that the plant language \( L \) is closed, i.e.,

\[ L = \overline{L} \]

Note that the plant may generate strings of events that cannot be extended to form a marked string. This phenomenon is called blocking. To clarify this issue, observe that for a plant model \((\Sigma, L, L_m)\) with closed \( L \), we always have the following relation (see Figure 4.1):

\[ L_m \subseteq \overline{L_m} \subseteq L \]

![Figure 4.1: Illustration of blocking phenomenon.](image)

\( L_m \) contains all marked strings, i.e., all strings that meet the terminal conditions (“have terminated successfully”). \( \overline{L_m} \setminus L_m \) contains all strings that have not terminated successfully yet, but can still be extended into a marked string. Finally, \( L \setminus \overline{L_m} \) contains all strings in \( L \) that cannot be extended into a marked string. The plant model \((\Sigma, L, L_m)\) is called non-blocking if \( L = L_m \), i.e., if no such strings exist.

### 4.3 Plant Controller Interaction

Before we discuss closed loop specifications and how to find a controller that will enforce them, we need to clarify the mode of interaction between plant and controller. For this, we assume that the controller is another DES defined on the same alphabet \( \Sigma \) as the plant but, of course, exhibiting different dynamics. The latter is captured by the controller language \( L_c \subseteq \Sigma^* \). We also assume that the marked language of the controller is identical to
its language. Therefore, the controller is completely described by

\[ C = (\Sigma, L_c, L_{cm} = L_c) \]  \hspace{1cm} (4.4)

As pointed out in the sequel, this implies that the controller will not change the marking properties introduced by the plant model. We will later realise the controller DES by a finite automaton. As the language generated by an automaton is always closed (see Section 4.5), we will henceforth also assume that \( L_c \) is closed, i.e., \( L_c = L_c \). It is obvious that the controller DES has to satisfy another (implementability) requirement. Namely, it can only disable events in the controllable subset \( \Sigma_c \) of \( \Sigma \):

**Definition 4.2 (Implementable controller)** The controller \( (4.4) \) is implementable for the plant model \( P \) if

\[ L_c \Sigma_{uc} \cap L \subseteq L_c, \]

where \( L_c \Sigma_{uc} := \{ s\sigma \mid s \in L_c, \sigma \in \Sigma_{uc}\} \).

This means that for any string \( s \in L_c \), if \( s \) is followed by an uncontrollable event \( \sigma \) and if the extended string \( s\sigma \) can be generated by the plant, \( s\sigma \) must also be a string in \( L_c \). In other words: an implementable controller accepts all uncontrollable events that the plant produces.

If the implementability requirement is satisfied, the interaction between plant and controller is simply to agree on strings that are both in \( L \) and in \( L_c \). Hence, the closed loop language is

\[ L_{cl} = L \cap L_c. \]

Similarly, a string of the closed loop system is marked if and only if it is marked by both the plant and the controller, i.e.,

\[ L_{cl,m} = L_m \cap L_c = L_m \cap L \cap L_c = L_m \cap L_{cl}. \]

Let us now rephrase our problem and ask which closed loop languages can be achieved by a controller satisfying the implementability constraints discussed above. The answer is not surprising:

**Theorem 4.1** There exists an implementable controller with closed language \( L_c \) such that

\[ L_c \cap L = K, \]  \hspace{1cm} (4.5)
4.4 Specifications

if and only if

(i) \( K \) is closed ,
(ii) \( K \subseteq L \),
(iii) \( K \Sigma_{uc} \cap L \subseteq K \).

(4.6)

Proof Sufficiency is straightforward, as (i)–(iii) imply that \( L_c = K \) is a suitable controller language: it is closed because of (i); because of (ii) it satisfies \( K \cap L = K \), and because of (iii) it is implementable for \( L \). Necessity of (i) and (ii) follows immediately from (4.5) and the fact that \( L_c \) and \( L \) are both closed languages. To show the necessity of (iii), assume that there exist \( s \in K \), \( \sigma \in \Sigma_{uc} \) such that \( s\sigma \in L \), \( s\sigma \notin K \), i.e., (iii) does not hold. Then, because of (4.5), \( s \in L_c \) and \( s\sigma \notin L_c \), i.e., the controller is not implementable for \( L \).

Remark 4.1 (4.6) is called the controllability condition for the closed language \( K \).

4.4 Specifications

The closed loop specifications are twofold:

(a) The closed loop language \( L_{cl} \) has to be a subset of a given specification language \( L_{spec} \), which is assumed to be closed:

\[
L_{cl} \subseteq L_{spec} \quad \text{with} \quad L_{spec} = \overline{L_{spec}} .
\]

(4.7)

It is therefore the task of control to prevent undesirable strings from occurring.

(b) The closed loop must be nonblocking, i.e.,

\[
\overline{L_{cl \cap m}} = L_{cl \cap m} \quad \text{with} \quad L_{cl \cap m} = L_{cl} .
\]

(4.8)

This means that any closed loop string must be extendable to form a marked string.

It is obvious that (4.7) implies

\[
L_{cl \cap m} = L_{cl} \cap L_{m} \subseteq L_{spec} \cap L_{m} .
\]

(4.9)

As the following argument shows, (4.8) and (4.9) also imply (4.7):

\[
L_{cl} = \overline{L_{cl \cap m}} \quad \text{(because of (4.8))}
\subseteq \overline{L_{spec} \cap L_{m}} \quad \text{(because of (4.9))}
\subseteq L_{spec} \cap L_{m} \quad \text{(always true)}
\subseteq \overline{L_{spec}} \quad \text{(always true)}
= L_{spec} \quad \text{(as } L_{spec} \text{ is closed)}.
\]
Instead of (4.7) and (4.8), we can therefore work with (4.8) and (4.9) as closed loop specifications. This, however, does not completely specify the closed loop. We therefore add the requirement that $L_{cl,m}$ should be as large as possible. In other words, we want control to be least restrictive or, equivalently, maximally permissive.

In summary, our control problem is to find an implementable controller

$$C = (\Sigma, L_c, L_c) ,$$

such that

1. the marked closed loop language satisfies (4.9)
2. the closed loop is nonblocking, i.e., (4.8) holds
3. control is maximally permissive.

This naturally leads to the question which nonblocking marked closed loop languages $K$ can be achieved by an implementable controller. The answer is provided by the following theorem:

**Theorem 4.2.** There exists an implementable controller with closed language $L_c$ such that

$$L_c \cap L_m = K \quad (4.10)$$

and

$$L_c \cap L = \overline{K} \quad (4.11)$$

if and only if

(i) $K \subseteq L_m ,$

(ii) $\overline{K}\Sigma_{uc} \cap L \subseteq \overline{K} ,$  \hspace{1cm} (4.12)

(iii) $K = \overline{K} \cap L_m . \hspace{1cm} (4.13)$

**Proof** Sufficiency is straightforward as (i)–(iii) imply that $L_c = \overline{K}$ is a suitable controller language: first, $L_c$ is obviously closed. Then, because of (iii), we have $L_c \cap L_m = \overline{K} \cap L_m = K,$ i.e., (4.10) holds. Furthermore, (i) and the fact that $L$ is closed implies $\overline{K} \subseteq L.$ Therefore, $L_{cl} = L_c \cap L = \overline{K} \cap L = \overline{K},$ i.e., (4.11) holds. Finally, (ii) says that $L_c = \overline{K}$ is implementable for $L.$

Necessity of (i) and (iii) follows directly from (4.10) and (4.11). To show necessity of (ii), assume that there exist $s \in \overline{K}, \sigma \in \Sigma_{uc}$ such that $s\sigma \in L, s\sigma \notin \overline{K},$ i.e., (iii) does not hold. Then, because of (4.11), $s \in L_c$ and $s\sigma \notin L_c,$ i.e., the controller is not implementable for $L.$ \hfill \blacksquare

**Remark 4.2.** (4.12) is called the controllability condition for $K,$ and (4.13) is known as the $L_m$-closedness condition.
Theorem 4.2 tells us whether we can achieve a nonblocking closed loop with a given marked language $K$. Recall that we want the maximal $K$ that satisfies $K \subseteq L_{spec} \cap L_m$. Hence we check whether

$$\hat{K} := L_{spec} \cap L_m \quad (4.14)$$

satisfies condition (ii) of Theorem 4.2. Note that (i) holds by definition for $\hat{K}$. As the following argument shows, (iii) also holds for $\hat{K}$:

$$\hat{K} = L_m \cap L_{spec}$$
$$= L_m \cap L_{spec} \cap L_m$$
$$\subseteq L_m \cap L_{spec} \cap L_m$$
$$= \overline{K} \cap L_m$$

and

$$\overline{K} \cap L_m = L_m \cap L_{spec} \cap L_m$$
$$\subseteq L_m \cap L_{spec} \cap L_m$$
$$= L_m \cap L_{spec}$$
$$= L_m \cap L_{spec}$$

as $L_{spec}$ is a closed language. Hence, if (ii) also holds, $\hat{K}$ is the desired maximally permissive marked closed loop language and $\overline{K}$ is a corresponding controller language. If the condition does not hold, we seek the least restrictive controllable sublanguage of $\hat{K}$, i.e.,

$$\hat{K} := \sup\{K \subseteq \hat{K} \mid (4.12) \text{ holds }\}.$$ 

Using set-theoretic arguments, it can be easily shown that $\hat{K}$ uniquely exists and is indeed controllable, i.e., satisfies Condition (ii) in Theorem 4.2. As $\hat{K} \subseteq \overline{K}$, (i) holds automatically. Furthermore, it can be shown (e.g., [3]) that $\hat{K}$ also satisfies (iii). Hence, $\hat{K}$ is the desired maximally permissive marked closed loop language and $\hat{K}$ is a suitable controller language.

Example 4.1 Consider the following exceedingly simple DES. Its purpose is to qualitatively model the water level in a reservoir. To do this, we introduce two threshold values for the (real-valued) level signal $x$, and four events:

$$\Sigma = \{o, \overline{o}, e, \overline{e}\}.$$ 

The event $o$ ("overflow") denotes that the water level crosses the upper threshold from below. The event $\overline{o}$ denotes that $x$ crosses this threshold from above. Similarly, $e$ ("empty") means that $x$
Supervisory Control

crosses the lower threshold from above, and \( \bar{x} \) that \( x \) crosses this threshold from below. We assume that initially the water level \( x \) is between the two thresholds, implying that the first event will either be \( o \) or \( e \). In our fictitious reservoir, we have no control over water consumption. The source for the reservoir is also unpredictable, but we can always close the pipe from the source to the reservoir (Figure 4.2) to shut down the feed.

![Diagram of water reservoir example]

This implies that \( o \) and \( \bar{e} \) are controllable events (they can be prohibited by control), whereas \( \bar{o} \) and \( e \) are not:

\[
\Sigma_c = \{ o, \bar{e} \}, \quad \Sigma_{uc} = \{ \bar{o}, e \}.
\]

The plant language is easily described in words: the first event is \( o \) or \( e \). After \( o \), only \( \bar{o} \) can occur. After \( e \), only \( \bar{e} \) can occur. After \( \bar{o} \) and \( \bar{e} \), either \( o \) or \( e \) may occur:

\[
L = \{ \varepsilon, o, e, o\bar{e}, e\bar{o}, o\bar{o}e, \ldots \} . \quad (4.15)
\]

Clearly, \( L \) is a closed language, i.e., \( L = \overline{L} \).

We consider those strings marked that correspond to a current value of \( x \) between the lower and upper threshold:

\[
L_m = \{ \varepsilon, o\bar{o}, e\bar{e}, \ldots \} , \quad (4.16)
\]

i.e., all strings that end with an \( \bar{o} \) or an \( \bar{e} \) event plus \( \varepsilon \), the string of length 0. To complete the example, suppose that the specification
requires that strings may not begin with $o\bar{e}$ (although this does not make any physical sense). Hence,

$$L_{spec} = \Sigma^* \setminus \{o\bar{e} \ldots\}, \quad (4.17)$$

and $L_{spec}$ is a closed language.

We can now, at least in principle, use the approach outlined in the previous pages to determine the least restrictive control strategy. First, we need to check whether $\hat{K} = L_m \cap L_{spec}$ can be achieved by means of an implementable controller. This is not possible, as condition (ii) in Theorem 4.2 is violated for $\hat{K}$. To see this, consider the string $o\bar{e}$. Clearly, $o\bar{e} \in L_m \cap L_{spec} = \hat{K}$. Therefore, $o\bar{e} \in \bar{K}\Sigma_{uc} \cap L$, but $o\bar{e} \notin \bar{K}$. Hence, (4.12) does not hold. This is also clear from Figure 4.3 which visualises the plant language $L$ as a tree.

![Figure 4.3: Illustration for Example 4.1](image)

From the figure, it is obvious that to enforce $\hat{K}$ as marked closed loop language, the controller would have to disable the event $e$ after the string $o\bar{e}$ has occurred. This is of course not possible as $e \in \Sigma_{uc}$. From the figure, it is also obvious what the least restrictive controllable sublanguage of $\hat{K}$ is: We need to prohibit that the first event is $o$ (by closing the pipe from the source to the reservoir). Once $e$ has occurred, $o$ can be enabled again. ♦

This example is meant to illustrate the basic idea in SCT. It also demonstrates, however, that we need a mechanism, i.e., a finite algorithm, to realise the required computations on the language level. This will be described in Section 4.5.

### 4.5 Controller Realisation

We first introduce finite automata as state models for both plant and specification. We then discuss a number of operations on
4. Supervisory Control

automata that will allow us to compute another finite automaton that realises the least restrictive controller.

4.5.1 Finite automata with marked states

**Definition 4.3 (Finite deterministic automaton)** A finite deterministic automaton with marked states is a quintuple

\[ \text{Aut} = (Q, \Sigma, f, q_0, Q_m) \] (4.18)

where \( Q \) is a finite set of states, \( \Sigma \) is a finite event set, \( f : Q \times \Sigma \rightarrow Q \) is a (partial) transition function, \( q_0 \in Q \) is the initial state, and \( Q_m \subseteq Q \) is the set of marked states.

To discuss the language and the marked language generated by \( \text{Aut} \), it is convenient to extend the transition function \( f : Q \times \Sigma \rightarrow Q \) to \( f : Q \times \Sigma^* \rightarrow Q \). This is done in a recursive way:

\[
\begin{align*}
    f(q, \varepsilon) &= q, \\
    f(q, s\sigma) &= f(f(q, s), \sigma) & \text{for } s \in \Sigma^* \text{ and } \sigma \in \Sigma.
\end{align*}
\]

Then, the language generated by \( \text{Aut} \) is

\[ L(\text{Aut}) := \{ s \in \Sigma^* \mid f(q_0, s) \text{ exists} \} . \]

The marked language generated by \( \text{Aut} \) (sometimes also called the language marked by \( \text{Aut} \)) is

\[ L_m(\text{Aut}) := \{ s \in \Sigma^* \mid f(q_0, s) \in Q_m \} . \]

Hence, \( L(\text{Aut}) \) is the set of strings that the automaton \( \text{Aut} \) can produce from its initial state \( q_0 \), and \( L_m(\text{Aut}) \) is the subset of strings that take the automaton from \( q_0 \) into a marked state. Clearly, the language generated by \( \text{Aut} \) is closed, i.e.

\[ L(\text{Aut}) = \overline{L(\text{Aut})} . \]

In general, this is not true for the language marked by \( \text{Aut} \), i.e.

\[ L_m(\text{Aut}) \subseteq \overline{L_m(\text{Aut})} . \]

We say that \( \text{Aut} \) realises the plant model \( P = (\Sigma, L, L_m) \) if

\[
\begin{align*}
    L(\text{Aut}) &= L, \\
    L_m(\text{Aut}) &= L_m .
\end{align*}
\]
Example 4.2 Let us reconsider the plant model from Example 4.1. The plant model \((\Sigma, L, L_m)\) is realised by \(Aut = (Q, \Sigma, f, q_0, Q_m)\) with

\[
Q = \{\text{Hi, Med, Lo}\},
q_0 = \text{Med},
Q_m = \{\text{Med}\},
\Sigma = \{o, \bar{e}, e, \bar{e}\},
\]

and \(f\) defined by the following table, where “–” means “undefined”.

<table>
<thead>
<tr>
<th></th>
<th>(o)</th>
<th>(\bar{e})</th>
<th>(e)</th>
<th>(\bar{e})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hi</td>
<td>–</td>
<td>Med</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Med</td>
<td>Hi</td>
<td>–</td>
<td>(\text{Lo})</td>
<td>–</td>
</tr>
<tr>
<td>Lo</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>Med</td>
</tr>
</tbody>
</table>

The resulting automaton is depicted in Figure 4.4. There, we use the following convention. The initial state is indicated by an arrow pointing “from the outside” to \(q_0\); marked states are indicated by arrows pointing from elements in \(Q_m\) “outside”; and controllable events can be recognised by a small bar added to the corresponding transition.

Figure 4.4: Automaton realisation for water reservoir system.

Clearly, 

\[
L(Aut) = \{\varepsilon, o, e, o\bar{e}, e\bar{e}, o\bar{o}, o\bar{e}, \ldots\},
\]

and 

\[
L_m(Aut) = \{\varepsilon, o\bar{e}, e\bar{e}, \ldots\}.
\]

\remark 4.3 A language that is marked by a finite deterministic automaton is called regular.
Remark 4.4. Aut is called non-blocking if the system \((\Sigma, L(Aut), L_m(Aut))\) is non-blocking, i.e., if
\[
L(Aut) = \overline{L_m(Aut)}.
\]
This implies that from any reachable state \(q\) of a non-blocking automaton, we can always get into a marked state. If in a blocking automaton, we get into a state \(q\) from which we cannot reach a marked state, we distinguish two situations: if there is no transition possible, i.e., if \(f(q, \sigma)\) is undefined \(\forall \sigma \in \Sigma\), we are in a deadlock situation, otherwise the automaton is said to be livelocked (Figure 4.5).

Figure 4.5: Deadlock (left) and livelock (right).

4.5.2 Unary operations on automata

We will need the following unary operations on automata, i.e., operations that take one finite deterministic automaton with marked states as an argument.
The first operation, \(Ac(Aut)\), removes all states that are not reachable (accessible) and all transitions originating from those states: for \(Aut\) given in (4.18),
\[
Ac(Aut) := (Q_{ac}, \Sigma, f_{ac}, q_0, Q_{ac,m}),
\]
where
\[
Q_{ac} := \{ q \in Q \mid \exists s \in \Sigma^* \text{ such that } f(q_0, s) = q \},
Q_{ac,m} := \{ q \in Q_m \mid \exists s \in \Sigma^* \text{ such that } f(q_0, s) = q \},
\]
\(f_{ac} : Q_{ac} \times \Sigma \to Q_{ac}\) is the restriction of \(f : Q \times \Sigma \to Q\) to \(Q_{ac}\).

Clearly, this operation neither changes the language nor the marked language generated by \(Aut\):
\[
L(Aut) = L(Ac(Aut))
\]
\[
L_m(Aut) = L_m(Ac(Aut)).
\]
Example 4.3 Consider the automaton depicted in the left part of Figure 4.6. Clearly, there is only one state that is not reachable. This state (and the two transitions originating from it) are removed by the Ac-operation to provide Ac(Aut) (right part of Figure 4.6).

\[\text{Figure 4.6: Illustration of Ac-operation.}\]

Another operation, CoAc(Aut), provides the “co-accessible” part of Aut. It removes all states from which we cannot reach a marked state and all transitions originating from and ending in such states. For Aut given in (4.18),

\[
\text{CoAc}(Aut) := (Q_{\text{coac}}, \Sigma, f_{\text{coac}}, \overline{q}_0, Q_m),
\]

where

\[
Q_{\text{coac}} := \{q \in Q \mid \exists s \in \Sigma^* \text{ such that } f(q, s) \in Q_m\}
\]

\[
\overline{q}_0 := \begin{cases} q_0, & \text{if } q_0 \in Q_{\text{coac}} \\ \text{undefined}, & \text{else} \end{cases}
\]

\[
f_{\text{coac}} : Q_{\text{coac}} \times \Sigma \rightarrow Q_{\text{coac}}
\]

is the restriction of \(f : Q \times \Sigma \rightarrow Q\) to \(Q_{\text{coac}}\).

Clearly, this operation does not change the language marked by Aut, i.e.,

\[L_m(Aut) = L_m(\text{CoAc}(Aut))\]

but will, in general, affect the language generated by Aut:

\[L(\text{CoAc}(Aut)) \subseteq L(Aut).\]

Note that, by construction, CoAc(Aut) is non-blocking, i.e.,

\[\overline{L}_m(\text{CoAc}(Aut)) = L(\text{CoAc}(Aut)).\]

Example 4.4 Consider the automaton depicted in the left part of Figure 4.7. Clearly, there are two states from which it is impossible to reach the marked state. These (plus the corresponding
transitions) are removed by the CoAc-operation to provide the nonblocking automaton shown in the right part of Figure 4.7.

**Remark 4.5** The Ac- and the CoAc-operation commute, i.e.,

\[ \text{Ac} (\text{CoAc}(\text{Aut})) = \text{CoAc} (\text{Ac}(\text{Aut})) \]

4.5.3 Binary operations on automata

The product operation, denoted by “×”, forces two automata to synchronise all events. For

\[
\begin{align*}
\text{Aut}_1 &= (Q_1, \Sigma, f_1, q_{10}, Q_{1m}) \\
\text{Aut}_2 &= (Q_2, \Sigma, f_2, q_{20}, Q_{2m}),
\end{align*}
\]

it is defined by

\[
\text{Aut}_1 \times \text{Aut}_2 := \text{Ac}(Q_1 \times Q_2, \Sigma, f, (q_{10}, q_{20}), Q_{1m} \times Q_{2m}) \tag{4.19}
\]

where \(Q_1 \times Q_2\) and \(Q_{1m} \times Q_{2m}\) denote Cartesian products, i.e., the sets of all ordered pairs from \(Q_1\) and \(Q_2\) and from \(Q_{1m}\) and \(Q_{2m}\), respectively. The transition function \(f\) of \(\text{Aut}_1 \times \text{Aut}_2\) is defined as follows:

\[
f((q_1, q_2), \sigma) = \begin{cases} 
(f_1(q_1, \sigma), f_2(q_2, \sigma)) & \text{if both } f_1(q_1, \sigma) \text{ and } f_2(q_2, \sigma) \text{ are defined}, \\
\text{undefined} & \text{else}.
\end{cases} \tag{4.20}
\]

Hence, in a state \((q_1, q_2)\) of the product automaton \(\text{Aut}_1 \times \text{Aut}_2\), an event \(\sigma \in \Sigma\) can only be generated if both \(\text{Aut}_1\) and \(\text{Aut}_2\) can generate \(\sigma\) in their respective states \(q_1\) and \(q_2\). In other words: the two constituent automata have to agree on, or synchronise, events. It follows from the definition (4.19) that the initial state
of $\text{Aut}_1 \times \text{Aut}_2$ is the pair of initial states of $\text{Aut}_1$ and $\text{Aut}_2$, and that a state $(q_1, q_2)$ is marked in $\text{Aut}_1 \times \text{Aut}_2$ if $q_1$ is marked in $\text{Aut}_1$ and $q_2$ is marked in $\text{Aut}_2$.

Note that, for convenience, we have included the $\text{Ac}$-operation into the product definition to remove non-reachable states.

The definition (4.19) implies the following properties:

$$L(\text{Aut}_1 \times \text{Aut}_2) = \{ s \in \Sigma^* | f(q_{10}, q_{20}), s \text{ exists} \}$$

$$= \{ s \in \Sigma^* | f_1(q_{10}, s) \text{ and } f_2(q_{20}, s) \text{ exist} \}$$

$$= \{ s \in \Sigma^* | f_1(q_{10}, s) \text{ exists} \} \cap \{ s \in \Sigma^* | f_2(q_{20}, s) \text{ exists} \}$$

$$= L(\text{Aut}_1) \cap L(\text{Aut}_2),$$

$$L_m(\text{Aut}_1 \times \text{Aut}_2) = \{ s \in \Sigma^* | f(q_{10}, q_{20}), s \in Q_m \}$$

$$= \{ s \in \Sigma^* | f_1(q_{10}, s) \in Q_{1m} \text{ and } f_2(q_{20}, s) \in Q_{2m} \}$$

$$= \{ s \in \Sigma^* | f_1(q_{10}, s) \in Q_{1m} \} \cap \{ s \in \Sigma^* | f_2(q_{20}, s) \in Q_{2m} \}$$

$$= L_m(\text{Aut}_1) \cap L_m(\text{Aut}_2).$$

Another operation on two automata is parallel composition, denoted by "∥". It is used to force synchronisation when the two constituent DESs (and therefore the two realising automata) are defined on different event sets. For

$$\text{Aut}_1 = (Q_1, \Sigma_1, f_1, q_{10}, Q_{1m})$$

and

$$\text{Aut}_2 = (Q_2, \Sigma_2, f_2, q_{20}, Q_{2m}),$$

$$\text{Aut}_1 ∥ \text{Aut}_2 := \text{Ac}(Q_1 \times Q_2, \Sigma_1 \cup \Sigma_2, f, (q_{10}, q_{20}), Q_{1m} \times Q_{2m}),$$

where

$$f((q_1, q_2), \sigma) = \begin{cases} (f_1(q_1, \sigma), f_2(q_2, \sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2, \text{ and both } f_1(q_1, \sigma) \text{ and } f_2(q_2, \sigma) \text{ are defined,} \\ (f_1(q_1, \sigma), q_2) & \text{if } \sigma \in \Sigma_1 \setminus \Sigma_2 \text{ and } f_1(q_1, \sigma) \text{ is defined,} \\ (q_1, f_2(q_2, \sigma)) & \text{if } \sigma \in \Sigma_2 \setminus \Sigma_1 \text{ and } f_2(q_2, \sigma) \text{ is defined,} \\ \text{undefined} & \text{else.} \end{cases}$$

(4.22)
This implies that the automata $Aut_1$ and $Aut_2$ only have to agree on events that are elements of both $\Sigma_1$ and $\Sigma_2$. Each automaton can generate an event without consent from the other automaton, if this event is not in the event set of the latter. In the special case where $\Sigma_1 \cap \Sigma_2 = \emptyset$, parallel composition is also called the “shuffle product”.

To discuss the effect of parallel composition on languages, we need to introduce projections. The projection operation

$$P_i : (\Sigma_1 \cup \Sigma_2)^* \rightarrow \Sigma_i^*, \ i = 1, 2,$$

is defined recursively as

$$P_i(\epsilon) = \epsilon$$

$$P_i(s\sigma) = \begin{cases} P_i(s)\sigma & \text{if } \sigma \in \Sigma_i, \\ P_i(s) & \text{otherwise}. \end{cases}$$

Hence, the effect of $P_i$ on a string $s \in (\Sigma_1 \cup \Sigma_2)^*$ is to remove all symbols that are not contained in $\Sigma_i$.

The inverse projection $P_i^{-1} : \Sigma_i^* \rightarrow 2^{(\Sigma_1 \cup \Sigma_2)^*}$ is defined as

$$P_i^{-1}(s) = \{ t \in (\Sigma_1 \cup \Sigma_2)^* \mid P_i(t) = s \}.$$ 

With these definitions, we can write

$$L(Aut_1 \parallel Aut_2) = \{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f((q_{10}, q_{20}), s) \text{ exists } \}$$

$$= \{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f_1(q_{10}, P_1(s)) \text{ and } f_2(q_{20}, P_2(s)) \text{ exist} \}$$

$$= \{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f_1(q_{10}, P_1(s)) \text{ exists} \} \cap$$

$$\{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f_2(q_{20}, P_2(s)) \text{ exists} \}$$

$$= P_1^{-1}(\{ t \in \Sigma_1^* \mid f_1(q_{10}, t) \text{ exists} \}) \cap$$

$$P_2^{-1}(\{ t \in \Sigma_2^* \mid f_2(q_{20}, t) \text{ exists} \})$$

$$= P_1^{-1}(L(Aut_1)) \cap P_2^{-1}(L(Aut_2)).$$

Similarly, we can show

$$L_m(Aut_1 \parallel Aut_2) = \{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f((q_{10}, q_{20}), s) \in Q_m \}$$

$$= \{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f_1(q_{10}, P_1(s)) \in Q_{1m} \}$$

and $f_2(q_{20}, P_2(s)) \in Q_{2m} \}$$

$$= \{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f_1(q_{10}, P_1(s)) \in Q_{1m} \} \cap$$

$$\{ s \in (\Sigma_1 \cup \Sigma_2)^* \mid f_2(q_{20}, P_2(s)) \in Q_{2m} \}$$

$$= P_1^{-1}(\{ t \in \Sigma_1^* \mid f_1(q_{10}, t) \in Q_{1m} \}) \cap$$

$$P_2^{-1}(\{ t \in \Sigma_2^* \mid f_2(q_{20}, t) \in Q_{2m} \})$$

$$= P_1^{-1}(L_m(Aut_1)) \cap P_2^{-1}(L_m(Aut_2)).$$

80
The parallel composition operation is particularly useful in the following scenario. Often, the specifications can be formulated in terms of a subset $\Sigma_{\text{spec}} \subseteq \Sigma$, i.e., $L_{\text{spec}} \subseteq \Sigma_{\text{spec}}^*$. Recall that a crucial step when computing the least restrictive controller is to perform the language intersection (4.14). As $L_{\text{spec}}$ and $L_m$ are now defined on different alphabets, we cannot directly intersect these languages. In this situation, we have two options:

(i) Use inverse projection

$$P^{-1}_{\text{spec}} : \Sigma_{\text{spec}}^* \to \Sigma^*$$

to introduce

$$L_{\text{spec}} = P^{-1}_{\text{spec}}(L_{\text{spec}}).$$

Then, $L_{\text{spec}} \cap L_m$ is well defined and can be computed by finding finite automata realisations

$$\text{Aut}_p = (Q_p, \Sigma, f_p, q_{p0}, Q_{pm})$$

for the plant model $(\Sigma, L, L_m)$ and

$$\text{Aut}_{\text{spec}} = (Q_{\text{spec}}, \Sigma, f_{\text{spec}}, q_{\text{spec}0}, Q_{\text{spec}})$$

for the specification $(\Sigma, L_{\text{spec}}, L_{\text{spec}})$, respectively. Then,

$$L_{\text{spec}} \cap L_m = L_m(\text{Aut}_p \parallel \text{Aut}_{\text{spec}}).$$

(ii) Alternatively, we can directly work with the language $L_{\text{spec}}$ and define an automaton realisation

$$\tilde{\text{Aut}}_{\text{spec}} = (\tilde{Q}_{\text{spec}}, \Sigma_{\text{spec}}, \tilde{f}_{\text{spec}}, \tilde{q}_{\text{spec}0}, \tilde{Q}_{\text{spec}}).$$

The desired language intersection is then generated by

$$L_m \cap P^{-1}_{\text{spec}}(L_{\text{spec}}) = L_m(\text{Aut}_p \parallel \tilde{\text{Aut}}_{\text{spec}}).$$

Clearly, this option is much more economical, as the number of transitions in $\tilde{\text{Aut}}_{\text{spec}}$ will in general be much less than in $\text{Aut}_{\text{spec}}$.

**Example 4.5** Let us reconsider the simple water reservoir from Example [4.1] with event set $\Sigma = \{o, \overline{o}, e, \overline{e}\}$. A finite automaton realisation

$$\text{Aut}_p = (Q_p, \Sigma, f_p, q_{p0}, Q_{pm}) \quad (4.23)$$

for the plant model has already been determined in Example [4.2]. Recall that the specification is that strings beginning with $o\overline{e}$ are not allowed, i.e., the specification language is

$$L_{\text{spec}} = \Sigma^* \setminus \{o\overline{e} \ldots\}. \quad (4.24)$$
We can easily find a finite automaton $\text{Aut}_{\text{spec}}$ generating $L_{\text{spec}}$. It is depicted in Figure 4.8 and works as follows: The state $\delta$ can be interpreted as a “safe state”. Once this is reached, all strings from $\Sigma^*$ are possible. Clearly, if the first event is not $o$, it can be followed by any string in $\Sigma^*$ without violating the specifications. Hence, $\bar{o}, e, \bar{e}$ will take us from the initial state $\alpha$ to the “safe state” $\delta$. If the first event is an $o$, this will take us to state $\beta$. There, we have to distinguish whether $\bar{o}$ occurs (this will result in a transition to $\gamma$), or any other event. In the latter case, violation of the specification is not possible any more, hence this takes us to the safe state $\delta$. Finally, in $\gamma$, anything is allowed apart from $e$. As the specification is not supposed to introduce any additional marking, we set $Q_{\text{spec},m} = Q_{\text{spec}} = \{\alpha, \beta, \gamma, \delta\}$.

The desired language intersection is then provided by

$$L_m \cap L_{\text{spec}} = L_m(\text{Aut}_p \times \text{Aut}_{\text{spec}}),$$  \hspace{1cm} (4.25)

and the product automaton $\text{Aut}_p \times \text{Aut}_{\text{spec}}$ is shown in Figure 4.9.

\[
\begin{array}{c}
\alpha \\
\beta \\
\gamma \\
\delta
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\beta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\delta \\
o \\
\bar{e}
\end{array}
\]

\[
\begin{array}{c}
o \\
\gamma \\
o \\
\bar{e}
\end{array}
\]
Note that we could also express our specification on the reduced event set $\Sigma_{\text{spec}} = \{o, e\}$. The specification language would then be

$$\tilde{L}_{\text{spec}} = \Sigma_{\text{spec}}^* \setminus \{oe\}. \quad (4.26)$$

An automaton realisation $\tilde{\text{Aut}}_{\text{spec}}$ for $\tilde{L}_{\text{spec}}$ is shown in Figure 4.10. The desired language intersection is now provided by

$$L_m \cap P_{\text{spec}}^{-1}(\tilde{L}_{\text{spec}}) = L_m(\text{Aut}_p \parallel \tilde{\text{Aut}}_{\text{spec}}), \quad (4.27)$$

and the parallel composition $\text{Aut}_p \parallel \tilde{\text{Aut}}_{\text{spec}}$ is shown in Figure 4.11.

4.5.4 Realising least restrictive implementable control

Recall that, on the basis of a finite automaton $\text{Aut}_p$ realising the plant model $P = (\Sigma, L, L_m)$ and a finite automaton $\tilde{\text{Aut}}_{\text{spec}}$ realising the specifications $(\Sigma, L_{\text{spec}}, L_{\text{spec}})$, or, equivalently, $\tilde{\text{Aut}}_{\text{spec}}$ realising $(\Sigma_{\text{spec}} \subseteq \Sigma, \tilde{L}_{\text{spec}}, \tilde{L}_{\text{spec}})$, we can compute

$$\text{Aut}_{ps} := \text{Aut}_p \times \text{Aut}_{\text{spec}}$$
$$= \text{Aut}_p \parallel \tilde{\text{Aut}}_{\text{spec}}$$
$$= (Q_{ps}, \Sigma_{ps}, f_{ps}, q_{ps0}, Q_{psm})$$
Supervisory Control

\[
\hat{K} = L_m(Aut_{ps}) = L_m \cap L_{spec} = L_m \cap P_{spec}(\bar{L}_{spec})
\]

(4.28)

as the potentially least restrictive marked closed loop language and \( \hat{K} \) as the potentially least restrictive closed loop (and controller) language. Note that a realisation of \((\Sigma, \hat{K}, \hat{K})\) is provided by

\[
Aut_{\hat{K}} := CoAc(Aut_{ps}) = (Q_{\hat{K}}, \Sigma, f_{\hat{K}}, q_{\hat{K}0}, Q_{\hat{K}m})
\]

as \( Aut_{ps} \) may be blocking.

We now need a mechanism to decide whether \( \hat{K} \) can be achieved by an implementable controller. If yes, \( \hat{K} = L_{\hat{K}}(Aut_{\hat{K}}) \) is the least restrictive (or maximally permissive) implementable controller. If not, we will need an algorithm to determine a realisation for the least restrictive controllable sublanguage \( \hat{K}^\uparrow \) of \( \hat{K} \).

We know that \( \hat{K} \) can be achieved by an implementable controller if and only if conditions \((i), (ii)\) and \((iii)\) in Theorem 4.2 hold for \( K = \hat{K} \). Because of the specific form (4.28) of the target language \( \hat{K} \), \((i)\) and \((iii)\) hold (see Section 4.4). Hence, we only need an algorithm to check condition \((ii)\) in Theorem 4.2. For this, introduce

\[
\Gamma_{\hat{K}}((q_1, q_2)) := \{ \sigma \in \Sigma \mid f_{\hat{K}}((q_1, q_2), \sigma) \text{ is defined} \}
\]

\[
\Gamma_p(q_1) := \{ \sigma \in \Sigma \mid f_p(q_1, \sigma) \text{ is defined} \}
\]

where \( f_{\hat{K}} \) and \( f_p \) are the transition functions of the automata \( Aut_{\hat{K}} \) and \( Aut_p \), respectively. Then, \((ii)\) holds for \( \hat{K} \) if and only if

\[
\Gamma_p(q_1) \setminus \Gamma_{\hat{K}}((q_1, q_2)) \subseteq \Sigma_c
\]

(4.29)

for all \((q_1, q_2) \in Q_{\hat{K}}\). If (4.29) is not true for some \((q_1, q_2) \in Q_{\hat{K}}\), this state and all the transitions originating in and ending in it are removed to give an automaton \( Aut_{\tilde{K}} \) with marked language

\[
\check{K} = L_m(Aut_{\tilde{K}})
\]

We apply the procedure consisting of CoAc- and Ac-operations\(^1\) and the subsequent removal of states that violate (4.29) recursively, until

\[
\Gamma_p(q_1) \setminus \Gamma_{\tilde{K}}((q_1, q_2)) \subseteq \Sigma_c
\]

\(1\) The Ac-operation can always be included, as it does neither affect the language nor the marked language.
holds for all \((q_1, q_2) \in \text{Aut}_K\). The resulting (non-blocking) automaton is \(\text{Aut}_K^\uparrow\), and its marked language is
\[
\hat{K}^\uparrow = L_m(\text{Aut}_K^\uparrow)
\]

**Example 4.6** We now apply this procedure to the automaton
\[
\text{Aut}_{ps} = (\text{Aut}_p \times \text{Aut}_{spec})
\]
from Example 4.5. As this \(\text{Aut}_{ps}\) is nonblocking, we have
\[
\text{Aut}_K^\uparrow = \text{Aut}_{ps}.
\]
Clearly, (4.29) does not hold for state \((\text{Med, } \gamma)\) in \(Q_K\). There,
\[
\Gamma_p(\text{Med}) = \{o, e\}
\]
\[
\Gamma_K(\text{Med}, \gamma) = \{o\}
\]
and therefore
\[
\Gamma_p(\text{Med}) \setminus \Gamma_K(\text{Med}, \gamma) = \{e\} \notin \Sigma_c.
\]
Removing this state (plus the corresponding transitions) provides \(\text{Aut}_K^\uparrow\) as shown in Figure 4.12. Applying the \(\text{CoAc}\)-operation

![Diagram](image)

**Figure 4.12: \(\text{Aut}_K^\uparrow\) for Example 4.6**

results in the automaton shown in Figure 4.13. Now (4.29) is satisfied for all \((q_1, q_2)\) in the state set of \(\text{CoAc}(\text{Aut}_K^\uparrow)\). Hence
\[
\text{Aut}_{K^\uparrow} = \text{CoAc}(\text{Aut}_K^\uparrow)
\]
is the desired controller realisation.

\[\diamondsuit\]

### 4.6 Control of a Manufacturing Cell

In this section, the main idea of SCT will be illustrated by means of a simple, but nontrivial, example. The example is adopted
from [8]. The manufacturing cell consists of two machines and an autonomous guided vehicle (AGV). Machine 1 can take a workpiece from a storage and do some preliminary processing. Before it can take another workpiece from the storage, it has to transfer the processed workpiece to the AGV. Machine 2 will then take the pre-processed workpiece from the AGV and add more processing steps. The finished workpiece then has again to be transferred to the AGV, which will finally deliver it to a conveyor belt. From a high-level point of view, we need the following events to describe the operation of the machines and the AGV.

The event set for machine 1 is $\Sigma_{M1} = \{M1T, M1P\}$, where $M1T$ signifies the event that a workpiece is being taken from the storage, and $M1P$ is the event that a workpiece is transferred from machine 1 to the AGV. $M1T$ is a controllable event, whereas $M1P$ is not controllable: if machine 1 is finished with a workpiece it will have to transfer it to the AGV. An automaton model for machine 1 is shown in Fig. 4.14.

The event set for machine 2 is $\Sigma_{M2} = \{M2T, M2P\}$, where $M2T$ represents the event that a preprocessed workpiece is transferred from the AGV to machine 2, and $M2P$ signifies that the finished workpiece is put from machine 2 to the AGV. As for machine 1, $M2T$ is a controllable event, whereas $M2P$ is not controllable. The automaton $M2$ (Fig. 4.15) models machine 2.
The event set for the AGV consists of four elements: \( \Sigma_{AGV} = \{M1P, M2T, M2P, CB\} \), where CB represents the event that a finished workpiece is being transferred from the AGV to the conveyor belt. CB is not controllable. We assume that the AGV has capacity one, i.e., it can only hold one workpiece at any instant of time. A suitable automaton model, \( VEH \), is shown in Figure 4.16.

In state \( \beta \), the AGV is not loaded; in state \( \alpha \), it is loaded with a preprocessed workpiece from machine 1; in \( \gamma \), it is loaded with a finished workpiece from machine 2.

In a first step, we set up the plant model by parallel composition of the three automata \( M1, M2, \) and \( VEH \). As \( \Sigma_{M1} \cap \Sigma_{M2} = \emptyset \), the parallel composition \( M := M1 \parallel M2 \) reduces to the “shuffle product”. This is shown in Figure 4.17, and \( Aut_p = M \parallel VEH \) is depicted in Figure 4.18.

Let’s first assume that the only requirement is that the closed loop is non-blocking, i.e., \( L_{spec} = (\Sigma_{M1} \cup \Sigma_{M2} \cup \Sigma_{AGV})^* \). It is indeed easy to see from Figure 4.18 that the uncontrolled plant, \( Aut_p \), may block. An example for a string of events that takes the plant state from its initial value into a blocking state is

\[
M1T, M1P, M1T, M2T, M1P, M1T.
\]

In the state reached by this string, both machines are loaded with workpieces, and the AGV is also loaded with a preprocessed workpiece, i.e., a workpiece which is not ready to be delivered to the conveyor belt.

---

Figure 4.15: Automaton model \( M2 \) for machine 2.

Figure 4.16: Automaton model \( VEH \) for the autonomous guided vehicle.
4 Supervisory Control

Figure 4.17: \( M = M_1 \parallel M_2 \).

Figure 4.18: Realisation of plant model, \( Aut_p = M \parallel VEH \).

Note that an automaton realisation \( Aut_{spec} \) for \( L_{spec} \) is trivial. Its state set is a singleton, and in this single state all events from \( \Sigma = \Sigma_{M_1} \cup \Sigma_{M_2} \cup \Sigma_{AGV} \) can occur. The first step in the controller synthesis procedure outlined in the previous section is to compute

\[
Aut_K = CoAc(Aut_p \times Aut_{spec}) = CoAc(Aut_p).
\]
This is shown in Figure 4.19. When investigating $\text{Aut}_K$, we find that (4.29) is violated in the state indicated by $\blacksquare$, as a transition corresponding to an uncontrollable event has been removed. Hence, we remove $\blacksquare$ (plus all transitions originating and ending there). This, however, gives rise to a blocking automaton. Applying the $\text{CoAc}$-operation for a second time results in the automaton shown in Figure 4.20. For this automaton, (4.29) is satisfied in all states; it is therefore the desired controller realisation $\text{Aut}_{K^?}$.

Let us assume that apart from non-blocking, we have another specification. Namely, it is required that each $M2P$ event is immediately followed by a $CB$ event. The corresponding specification can be realised by the automaton $\text{Aut}_{\text{spec}}$ shown in Figure 4.21. The corresponding automaton $\text{Aut}_{ps} = \text{Aut}_p \times \text{Aut}_{\text{spec}}$ is depicted in Figure 4.22. We then compute $\text{Aut}_K = \text{CoAc}(\text{Aut}_p \times \text{Aut}_{\text{spec}})$ and perform the discussed controller synthesis procedure. The resulting $\text{Aut}_{K^?}$ is shown in Figure 4.23.
Figure 4.20: Realisation of least restrictive controller $\text{Aut}_{k^1}$.

Figure 4.21: Specification automaton $\text{Aut}_{\text{spec}}$. 

4 Supervisory Control
4.6 Control of a Manufacturing Cell

Figure 4.22: \( Aut_p \times Aut_{spec} \).

Figure 4.23: Realisation of least restrictive controller \( Aut_{\mathcal{K}} \).
4 Supervisory Control
BIBLIOGRAPHY


